# Genius Manual

Copyright © 1997-2014 Jiří (George) Lebl

Copyright © 2004 Kai Willadsen

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License (GFDL), Version 1.1 or any later version published by the Free Software Foundation with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. You can find a copy of the GFDL at this link or in the file COPYING-DOCS distributed with this manual.

This manual is part of a collection of GNOME manuals distributed under the GFDL. If you want to distribute this manual separately from the collection, you can do so by adding a copy of the license to the manual, as described in section 6 of the license.

Many of the names used by companies to distinguish their products and services are claimed as trademarks. Where those names appear in any GNOME documentation, and the members of the GNOME Documentation Project are made aware of those trademarks, then the names are in capital letters or initial capital letters.

DOCUMENT AND MODIFIED VERSIONS OF THE DOCUMENT ARE PROVIDED UNDER THE TERMS OF THE GNU FREE DOCUMENTATION LICENSE WITH THE FURTHER UNDERSTANDING THAT:

1. DOCUMENT IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT IS FREE OF DEFECTS MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY, ACCURACY, AND PERFORMANCE OF THE DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT IS WITH YOU. SHOULD ANY DOCUMENT OR MODIFIED VERSION PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL WRITER, AUTHOR OR ANY CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER; AND
2. UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER IN TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL THE AUTHOR, INITIAL WRITER, ANY CONTRIBUTOR, OR ANY DISTRIBUTOR OF THE DOCUMENT OR MODIFIED VERSION OF THE DOCUMENT, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER DAMAGES OR LOSSES ARISING OUT OF OR RELATING TO USE OF THE DOCUMENT AND MODIFIED VERSIONS OF THE DOCUMENT, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES.

**Feedback**

To report a bug or make a suggestion regarding the Genius Mathematics Tool application or this manual, please visit the Genius Web page or email me at jirka@5z.com.

**COLLABORATORS**

| | *TITLE* :<br><br>Genius Manual | |
|---|---|---|

| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
|---|---|---|---|
| WRITTEN BY | Jiří Lebl and Kai Willadsen | March 9, 2015 | |

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| 0.2 | October 2014 | Jiri (George) Lebl jirka@5z.com | |

# Contents

**1 Introduction** **1**

**2 Getting Started** **2**

2.1 To Start Genius Mathematics Tool . . . 2

2.2 When You Start Genius . . . 2

**3 Basic Usage** **5**

3.1 Using the Work Area . . . 5

3.2 To Create a New Program . . . 6

3.3 To Open and Run a Program . . . 6

**4 Plotting** **7**

4.1 Line Plots . . . 7

4.2 Parametric Plots . . . 9

4.3 Slopefield Plots . . . 11

4.4 Vectorfield Plots . . . 12

4.5 Surface Plots . . . 12

**5 GEL Basics** **14**

5.1 Values . . . 14

5.1.1 Numbers . . . 14

5.1.2 Booleans . . . 15

5.1.3 Strings . . . 15

5.1.4 Null . . . 16

5.2 Using Variables . . . 16

5.2.1 Setting Variables . . . 17

5.2.2 Built-in Variables . . . 17

5.2.3 Previous Result Variable . . . 17

5.3 Using Functions . . . 17

5.3.1 Defining Functions . . . 18

5.3.2 Variable Argument Lists . . . 18

5.3.3 Passing Functions to Functions . . . 18
5.3.4 Operations on Functions . . . 19
5.4 Separator . . . 19
5.5 Comments . . . 20
5.6 Modular Evaluation . . . 20
5.7 List of GEL Operators . . . 20

**6 Programming with GEL** **24**
6.1 Conditionals . . . 24
6.2 Loops . . . 24
6.2.1 While Loops . . . 24
6.2.2 For Loops . . . 25
6.2.3 Foreach Loops . . . 25
6.2.4 Break and Continue . . . 25
6.3 Sums and Products . . . 26
6.4 Comparison Operators . . . 26
6.5 Global Variables and Scope of Variables . . . 26
6.6 Parameter variables . . . 27
6.7 Returning . . . 28
6.8 References . . . 28
6.9 Lvalues . . . 29

**7 Advanced Programming with GEL** **30**
7.1 Error Handling . . . 30
7.2 Toplevel Syntax . . . 30
7.3 Returning Functions . . . 31
7.4 True Local Variables . . . 32
7.5 GEL Startup Procedure . . . 32
7.6 Loading Programs . . . 33

**8 Matrices in GEL** **34**
8.1 Entering Matrices . . . 34
8.2 Conjugate Transpose and Transpose Operator . . . 35
8.3 Linear Algebra . . . 35

**9 Polynomials in GEL** **36**
9.1 Using Polynomials . . . 36

**10 Set Theory in GEL** **37**
10.1 Using Sets . . . 37

**11 List of GEL functions** **38**
11.1 Commands . . . 38
11.2 Basic . . . 38
11.3 Parameters . . . 42
11.4 Constants . . . 45
11.5 Numeric . . . 45
11.6 Trigonometry . . . 49
11.7 Number Theory . . . 51
11.8 Matrix Manipulation . . . 55
11.9 Linear Algebra . . . 58
11.10 Combinatorics . . . 65
11.11 Calculus . . . 67
11.12 Functions . . . 69
11.13 Equation Solving . . . 72
11.14 Statistics . . . 74
11.15 Polynomials . . . 75
11.16 Set Theory . . . 76
11.17 Commutative Algebra . . . 77
11.18 Miscellaneous . . . 77
11.19 Symbolic Operations . . . 77
11.20 Plotting . . . 78

**12 Example Programs in GEL** **84**

**13 Settings** **86**
13.1 Output . . . 86
13.2 Precision . . . 87
13.3 Terminal . . . 87
13.4 Memory . . . 87

**14 About Genius Mathematics Tool** **88**

# List of Figures

2.1 Genius Mathematics Tool Window . . . . . 3

4.1 Create Plot Window . . . . . 8
4.2 Plot Window . . . . . 9
4.3 Parametric Plot Tab . . . . . 10
4.4 Parametric Plot . . . . . 11
4.5 Surface Plot . . . . . 13

**Abstract**

Manual for the Genius Math Tool.

# Chapter 1

# Introduction

The Genius Mathematics Tool application is a general calculator for use as a desktop calculator, an educational tool in mathematics, and is useful even for research. The language used in Genius Mathematics Tool is designed to be 'mathematical' in the sense that it should be 'what you mean is what you get'. Of course that is not an entirely attainable goal. Genius Mathematics Tool features rationals, arbitrary precision integers and multiple precision floats using the GMP library. It handles complex numbers using cartesian notation. It has good vector and matrix manipulation and can handle basic linear algebra. The programming language allows user defined functions, variables and modification of parameters.

Genius Mathematics Tool comes in two versions. One version is the graphical GNOME version, which features an IDE style interface and the ability to plot functions of one or two variables. The command line version does not require GNOME, but of course does not implement any feature that requires the graphical interface.

Parts of this manual describe the graphical version of the calculator, but the language is of course the same. The command line only version lacks the graphing capabilities and all other capabilities that require the graphical user interface.

Generally, when some feature of the language (function, operator, etc...) is new is some version past 1.0.5, it is mentioned, but below 1.0.5 you would have to look at the NEWS file.

# Chapter 2

# Getting Started

## 2.1 To Start Genius Mathematics Tool

You can start Genius Mathematics Tool in the following ways:

**Applications menu** Depending on your operating system and version, the menu item for Genius Mathematics Tool could appear in a number of different places. It can be in the Education, Accessories, Office, Science, or similar submenu, depending on your particular setup. The menu item name you are looking for is Genius Math Tool. Once you locate this menu item click on it to start Genius Mathematics Tool.

**Run dialog** Depending on your system installation the menu item may not be available. If it is not, you can open the Run dialog and execute **gnome-genius**.

**Command line** To start the GNOME version of Genius Mathematics Tool execute **gnome-genius** from the command line.

To start the command line only version, execute the following command: **genius**. This version does not include the graphical environment and some functionality such as plotting will not be available.

## 2.2 When You Start Genius

When you start the GNOME edition of Genius Mathematics Tool, the window pictured in Figure 2.1 is displayed.

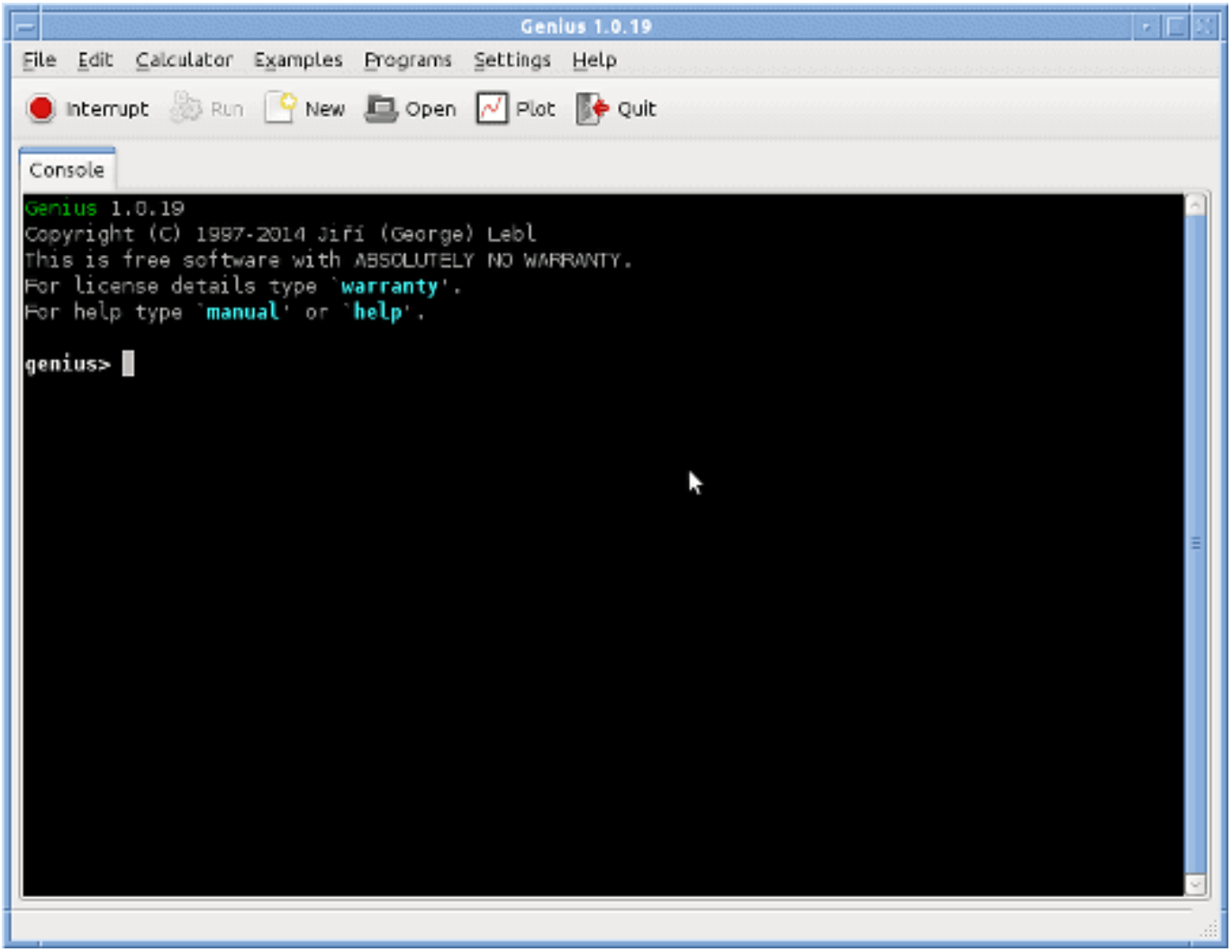


Figure 2.1: Genius Mathematics Tool Window

The Genius Mathematics Tool window contains the following elements:

**Menubar.** The menus on the menubar contain all of the commands that you need to work with files in Genius Mathematics Tool. The File menu contains items for loading and saving items and creating new programs. The Load and Run... command does not open a new window for the program, but just executes the program directly. It is equivalent to the **load** command.

The Calculator menu controls the calculator engine. It allows you to run the currently selected program or to interrupt the current calculation. You can also look at the full expression of the last answer (useful if the last answer was too large to fit onto the console), or you can view a listing of the values of all user defined variables. You can also monitor user variables, which is especially useful while a long calculation is running, or to debug a certain program. Finally the Calculator allows plotting functions using a user friendly dialog box.

The Examples menu is a list of example programs or demos. If you open the menu, it will load the example into a new program, which you can run, edit, modify, and save. These programs should be well documented and generally demonstrate some either feature of Genius Mathematics Tool or some mathematical concept.

The Programs menu lists the currently open programs and allows you to switch between them.

The other menus have same familiar functions as in other applications.

**Toolbar.** The toolbar contains a subset of the commands that you can access from the menubar.

**Working area** The working area is the primary method of interacting with the application.

The working area initially has just the Console tab, which is the main way of interacting with the calculator. Here you type expressions and the results are immediately returned after you hit the Enter key.

Alternatively you can write longer programs and those can appear in separate tabs. The programs are a set of commands or functions that can be run all at once rather than entering them at the command line. The programs can be saved in files for later retrieval.

# Chapter 3

# Basic Usage

## 3.1 Using the Work Area

Normally you interact with the calculator in the Console tab of the work area. If you are running the text only version then the console will be the only thing that is available to you. If you want to use Genius Mathematics Tool as a calculator only, just type in your expression in the console, it will be evaluated, and the returned value will be printed.

To evaluate an expression, type it into the Console work area and press enter. Expressions are written in a language called GEL. The most simple GEL expression just looks like mathematics. For example

```
genius> 30*70 + 67^3.0 + ln(7) * (88.8/100)
```

or

```
genius> 62734 + 812634 + 77^4 mod 5
```

or

```
genius> | sin(37) - e^7 |
```

or

```
genius> sum n=1 to 70 do 1/n
```

(Last is the harmonic sum from 1 to 70)

To get a list of functions and commands, type:

```
genius> help
```

If you wish to get more help on a specific function, type:

```
genius> help FunctionName
```

To view this manual, type:

```
genius> manual
```

Suppose you have previously saved some GEL commands as a program to a file and you now want to execute them. To load this program from the file `path/to/program.gel`, type

```
genius> load path/to/program.gel
```

Genius Mathematics Tool keeps track of the current directory. To list files in the current directory type **ls**, to change directory do **`cd directory`** as in the UNIX command shell.

## 3.2 To Create a New Program

If you wish to enter several more complicated commands, or perhaps write a complicated function using the GEL language. You can create a new program.

To start writing a new program, choose File → New Program. A new tab will appear in the work area. You can write a GEL program in this work area. Once you have written your program you can run it by Calculator → Run (or the Run toolbar button). This will execute your program and will display any output on the Console tab. Executing a program is equivalent of taking the text of the program and typing it into the console. The only difference is that this input is done independent of the console and just the output goes onto the console. Calculator → Run will always run the currently selected program even if you are on the Console tab. The currently selected program has its tab in bold type. To select a program, just click on its tab.

To save the program you've just written, choose File → Save As.... Similarly as in other programs you can choose File → Save to save a program that already has a filename attached to it. If you have many opened programs you have edited and wish to save you can also choose File → Save All Unsaved.

Programs that have unsaved changes will have a "[+]" next to their filename. This way you can see if the file on disk and the currently opened tab differ in content. Programs which have not yet had a filename associated with them are always considered unsaved and no "[+]" is printed.

## 3.3 To Open and Run a Program

To open a file, choose File → Open. A new tab containing the file will appear in the work area. You can use this to edit the file.

To run a program from a file, choose File → Load and Run.... This will run the program without opening it in a separate tab. This is equivalent to the **load** command.

If you have made edits to a file you wish to throw away and want to reload to the version that's on disk, you can choose the File → Reload from Disk menuitem. This is useful for experimenting with a program and making temporary edits, to run a program, but that you do not intend to keep.

# Chapter 4

# Plotting

Plotting support is only available in the graphical GNOME version. All plotting accessible from the graphical interface is available from the Create Plot window. You can access this window by either clicking on the Plot button on the toolbar or selecting Plot from the Calculator menu. You can also access the plotting functionality by using the plotting functions of the GEL language. See Chapter 5 to find out how to enter expressions that Genius understands.

## 4.1 Line Plots

To graph real valued functions of one variable open the Create Plot window. You can also use the LinePlot function on the command line (see its documentation).

Once you click the Plot button, a window opens up with some notebooks in it. You want to be in the Function line plot notebook tab, and inside you want to be on the Functions / Expressions notebook tab. See Figure 4.1.

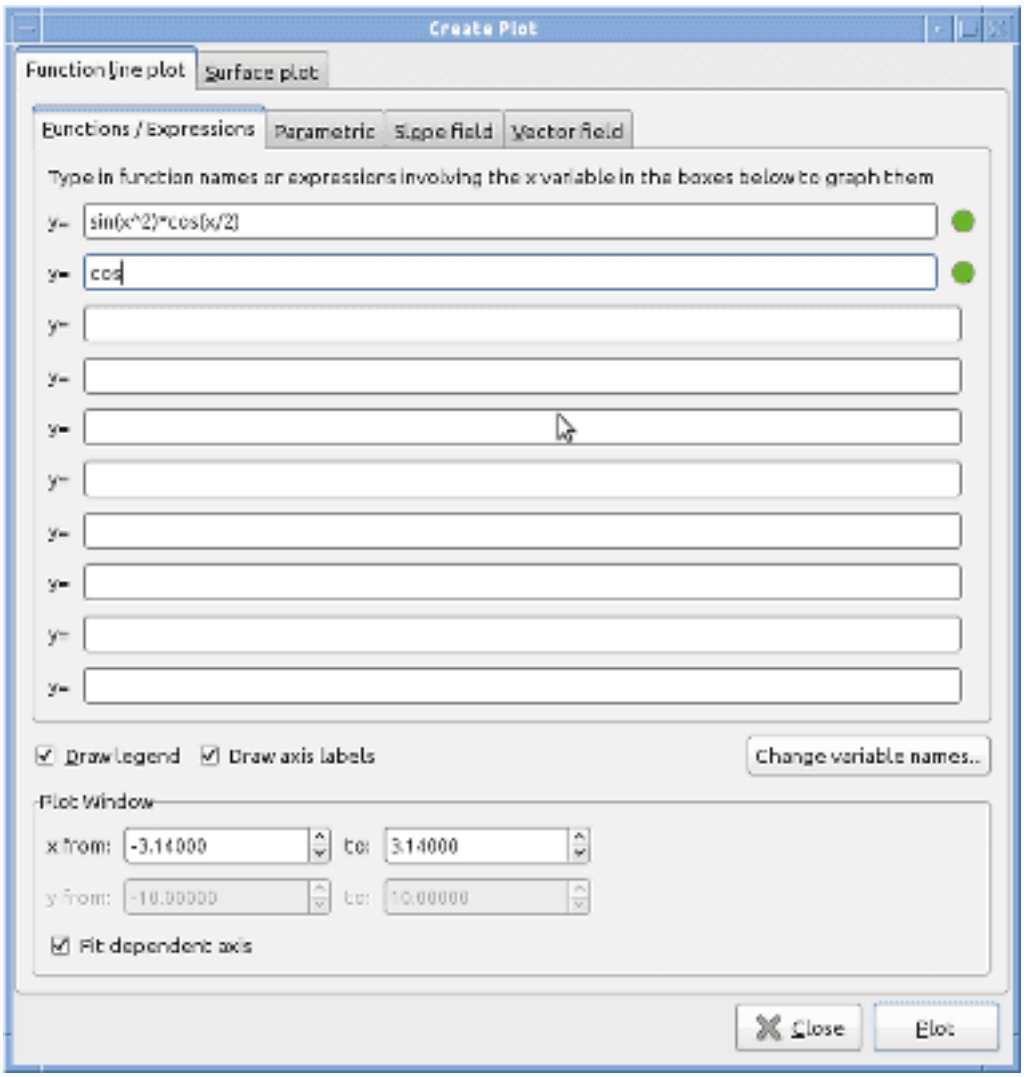


Figure 4.1: Create Plot Window

Type expressions with **x** as the independent variable into the textboxes. Alternatively you can give names of functions such as **cos** rather then having to type **cos(x)** . You can graph up to ten functions. If you make a mistake and Genius cannot parse the input it will signify this with a warning icon on the right of the text input box where the error occurred, as well as giving you an error dialog. You can change the ranges of the dependent and independent variables in the bottom part of the dialog. The `y` (dependent) range can be set automatically by turning on the Fit dependent axis checkbox. The names of the variables can also be changed. Pressing the Plot button produces the graph shown in Figure 4.2.

The variables can be renamed by clicking the Change variable names... button, which is useful if you wish to print or save the figure and don't want to use the standard names. Finally you can also avoid printing the legend and the axis labels completely, which is also useful if printing or saving, when the legend might simply be clutter.

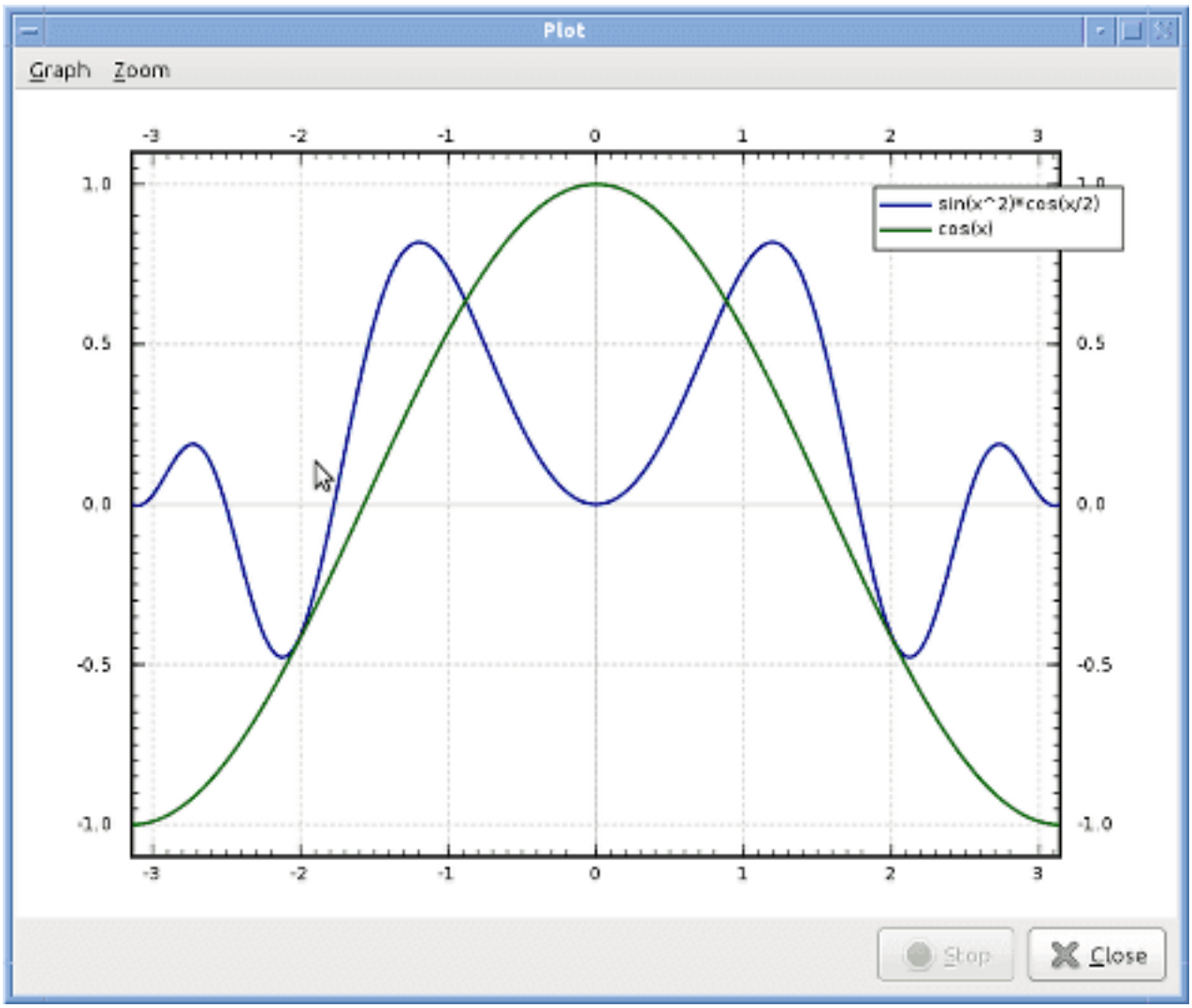


Figure 4.2: Plot Window

From here you can print out the plot, create encapsulated postscript or a PNG version of the plot or change the zoom. If the dependent axis was not set correctly you can have Genius fit it by finding out the extrema of the graphed functions.

For plotting using the command line see the documentation of the LinePlot function.

## 4.2 Parametric Plots

In the create plot window, you can also choose the Parametric notebook tab to create two dimensional parametric plots. This way you can plot a single parametric function. You can either specify the points as `x` and `y`, or giving a single complex number as a function of the variable `t`. The range of the variable `t` is given explicitly, and the function is sampled according to the given increment. The `x` and `y` range can be set automatically by turning on the Fit dependent axis checkbox, or it can be specified explicitly. See Figure 4.3.

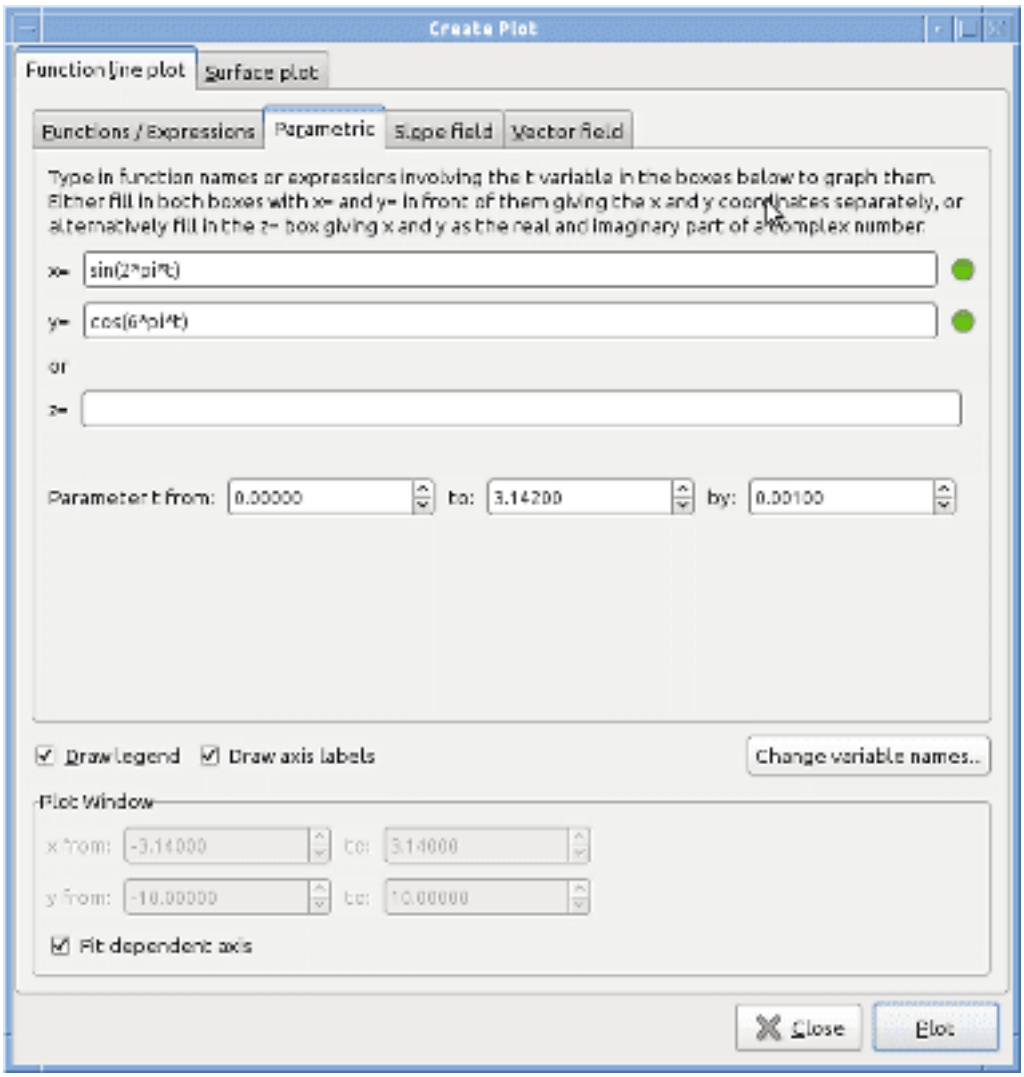


Figure 4.3: Parametric Plot Tab

An example of a parametric plot is given in Figure 4.4. Similar operations can be done on such graphs as can be done on the other line plots. For plotting using the command line see the documentation of the LinePlotParametric or LinePlotCParametric function.

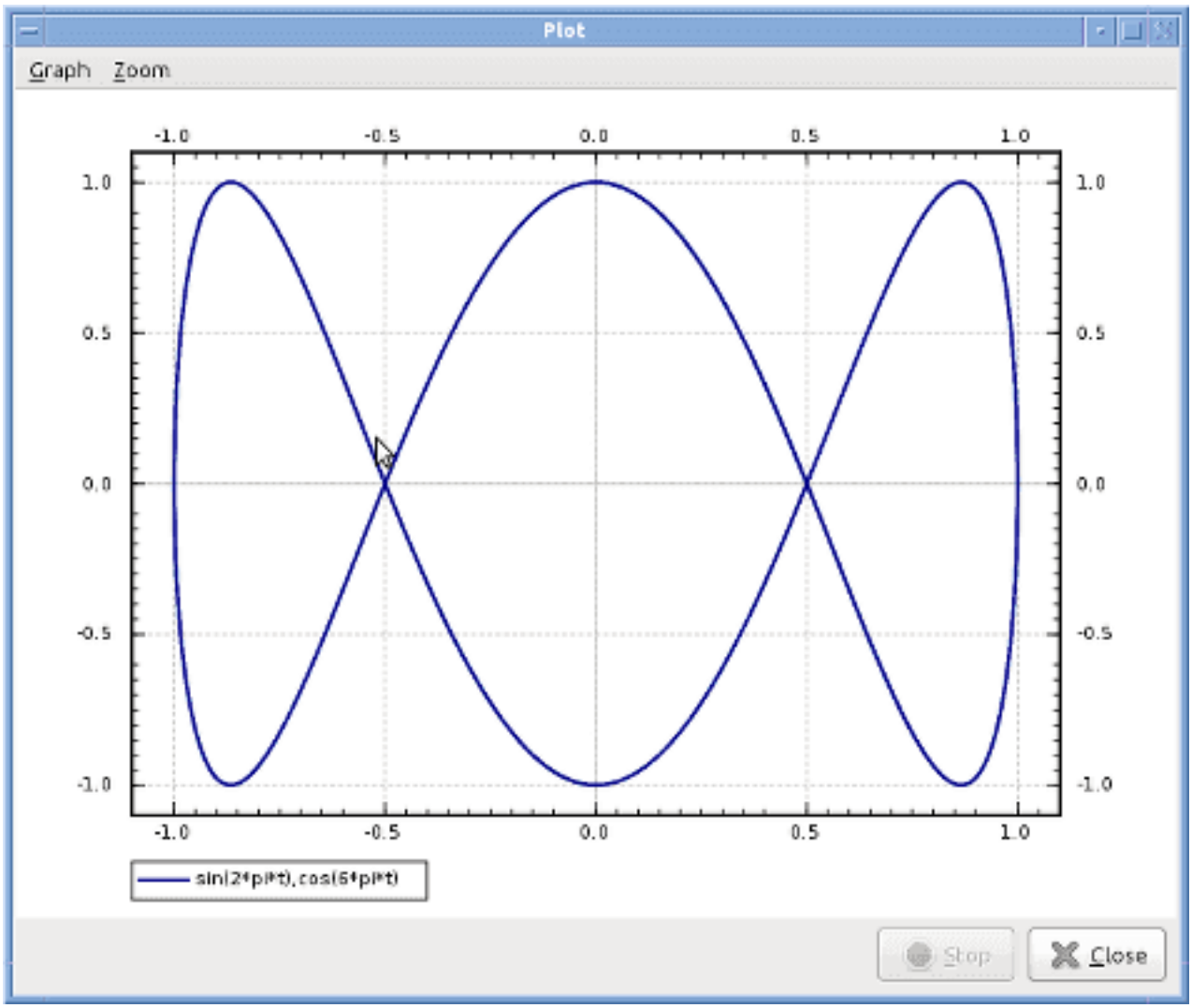


Figure 4.4: Parametric Plot

## 4.3 Slopefield Plots

In the create plot window, you can also choose the Slope field notebook tab to create a two dimensional slope field plot. Similar operations can be done on such graphs as can be done on the other line plots. For plotting using the command line see the documentation of the SlopefieldPlot function.

When a slope field is active, there is an extra Solver menu available, through which you can bring up the solver dialog. Here you can have Genius plot specific solutions for the given initial conditions. You can either specify initial conditions in the dialog, or you can click on the plot directly to specify the initial point. While the solver dialog is active, the zooming by clicking and dragging does not work. You have to close the dialog first if you want to zoom using the mouse.

The solver uses the standard Runge-Kutta method. The plots will stay on the screen until cleared. The solver will stop whenever it reaches the boundary of the plot window. Zooming does not change the limits or parameters of the solutions, you will have to clear and redraw them with appropriate parameters. You can also use the SlopefieldDrawSolution function to draw solutions from the command line or programs.

## 4.4 Vectorfield Plots

In the create plot window, you can also choose the Vector field notebook tab to create a two dimensional vector field plot. Similar operations can be done on such graphs as can be done on the other line plots. For plotting using the command line see the documentation of the VectorfieldPlot function.

By default the direction and magnitude of the vector field is shown. To only show direction and not the magnitude, check the appropriate checkbox to normalize the arrow lengths.

When a vector field is active, there is an extra Solver menu available, through which you can bring up the solver dialog. Here you can have Genius plot specific solutions for the given initial conditions. You can either specify initial conditions in the dialog, or you can click on the plot directly to specify the initial point. While the solver dialog is active, the zooming by clicking and dragging does not work. You have to close the dialog first if you want to zoom using the mouse.

The solver uses the standard Runge-Kutta method. The plots will stay on the screen until cleared. Zooming does not change the limits or parameters of the solutions, you will have to clear and redraw them with appropriate parameters. You can also use the VectorfieldDrawSolution function to draw solutions from the command line or programs.

## 4.5 Surface Plots

Genius can also plot surfaces. Select the Surface plot tab in the main notebook of the Create Plot window. Here you can specify a single expression that should use either `x` and `y` as real independent variables or `z` as a complex variable (where `x` is the real part of `z` and `y` is the imaginary part). For example to plot the modulus of the cosine function for complex parameters, you could enter `|cos(z)|`. This would be equivalent to `|cos(x+1i*y)|`. See Figure 4.5. For plotting using the command line see the documentation of the SurfacePlot function.

The `z` range can be set automatically by turning on the Fit dependent axis checkbox. The variables can be renamed by clicking the Change variable names... button, which is useful if you wish to print or save the figure and don't want to use the standard names. Finally you can also avoid printing the legend, which is also useful if printing or saving, when the legend might simply be clutter.

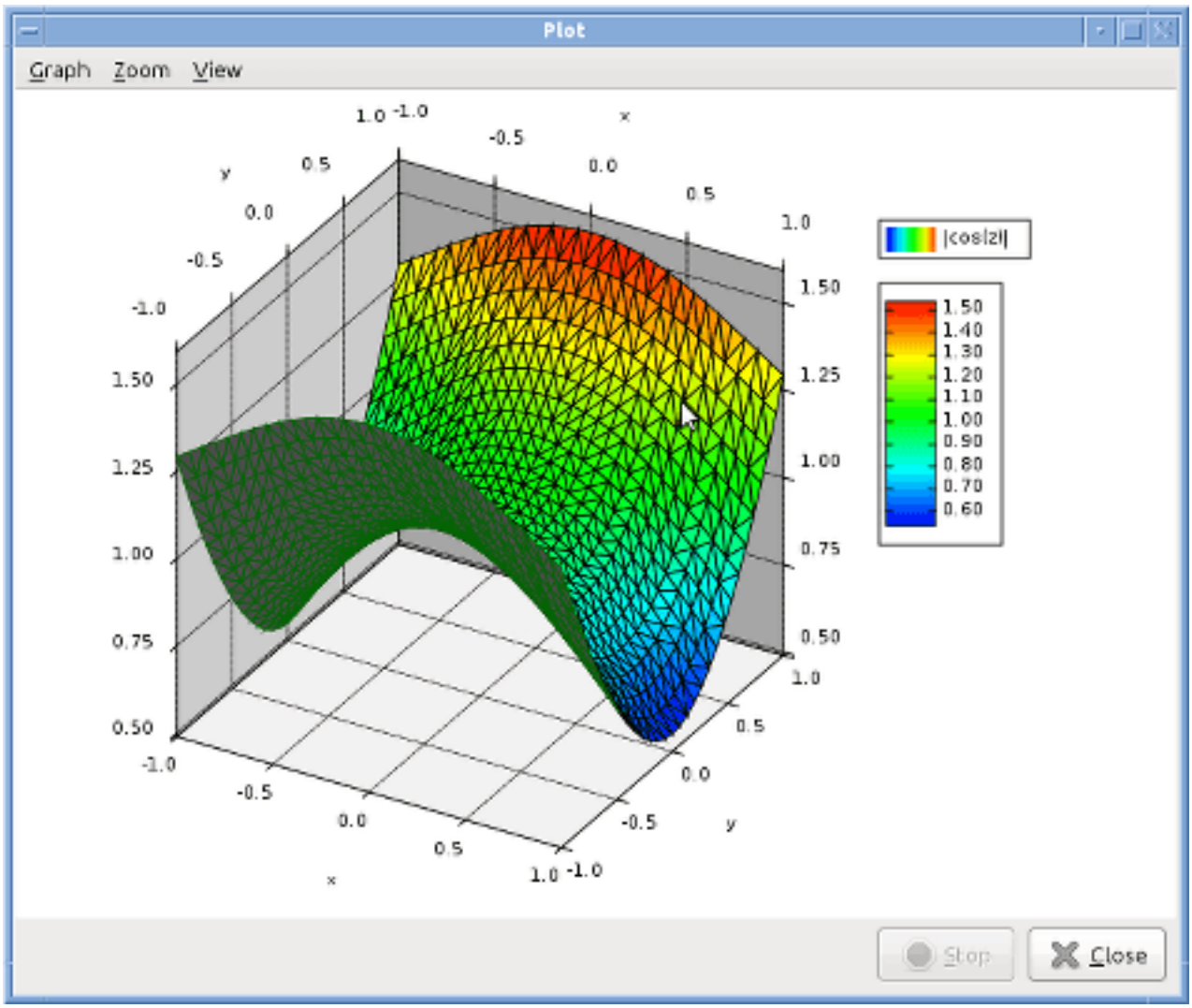


Figure 4.5: Surface Plot

In surface mode, left and right arrow keys on your keyboard will rotate the view along the z axis. Alternatively you can rotate along any axis by selecting Rotate axis... in the View menu. The View menu also has a top view mode which rotates the graph so that the z axis is facing straight out, that is, we view the graph from the top and get essentially just the colors that define the values of the function getting a temperature plot of the function. Finally you should try Start rotate animation, to start a continuous slow rotation. This is especially good if using Genius Mathematics Tool to present to an audience.

# Chapter 5

# GEL Basics

GEL stands for Genius Extension Language. It is the language you use to write programs in Genius. A program in GEL is simply an expression that evaluates to a number, a matrix, or another object in GEL. Genius Mathematics Tool can be used as a simple calculator, or as a powerful theoretical research tool. The syntax is meant to have as shallow of a learning curve as possible, especially for use as a calculator.

## 5.1 Values

Values in GEL can be numbers, Booleans, or strings. GEL also treats matrices as values. Values can be used in calculations, assigned to variables and returned from functions, among other uses.

### 5.1.1 Numbers

Integers are the first type of number in GEL. Integers are written in the normal way.

```
1234
```

Hexadecimal and octal numbers can be written using C notation. For example:

```
0x123ABC
01234
```

Or you can type numbers in an arbitrary base using `<base>\<number>`. Digits higher than 10 use letters in a similar way to hexadecimal. For example, a number in base 23 could be written:

```
23\1234ABCD
```

The second type of GEL number is rationals. Rationals are simply achieved by dividing two integers. So one could write:

```
3/4
```

to get three quarters. Rationals also accept mixed fraction notation. So in order to get one and three tenths you could write:

```
1 3/10
```

The next type if number is floating point. These are entered in a similar fashion to C notation. You can use `E`, `e` or `@` as the exponent delimiter. Note that using the exponent delimiter gives a float even if there is no decimal point in the number. Examples:

```
1.315
7.887e77
7.887e-77
.3
0.3
77e5
```

When Genius prints a floating point number it will always append a `.0` even if the number is whole. This is to indicate that floating point numbers are taken as imprecise quantities. When a number is written in the scientific notation, it is always a floating point number and thus Genius does not print the `.0`.

The final type of number in gel is the complex numbers. You can enter a complex number as a sum of real and imaginary parts. To add an imaginary part, append an `i`. Here are examples of entering complex numbers:

```
1+2i
8.01i
77*e^(1.3i)
```

**Important**
When entering imaginary numbers, a number must be in front of the `i`. If you use `i` by itself, Genius will interpret this as referring to the variable `i`. If you need to refer to `i` by itself, use `1i` instead.
In order to use mixed fraction notation with imaginary numbers you must have the mixed fraction in parentheses. (i.e., **`(1 2/5)i`**)

### 5.1.2 Booleans

Genius also supports native Boolean values. The two Boolean constants are defined as `true` and `false`; these identifiers can be used like any other variable. You can also use the identifiers `True`, `TRUE`, `False` and `FALSE` as aliases for the above.

At any place where a Boolean expression is expected, you can use a Boolean value or any expression that produces either a number or a Boolean. If Genius needs to evaluate a number as a Boolean it will interpret 0 as `false` and any other number as `true`.

In addition, you can do arithmetic with Boolean values. For example:

```
( (1 + true) - false ) * true
```

is the same as:

```
( (true or true) or not false ) and true
```

Only addition, subtraction and multiplication are supported. If you mix numbers with Booleans in an expression then the numbers are converted to Booleans as described above. This means that, for example:

```
1 == true
```

always evaluates to `true` since 1 will be converted to `true` before being compared to `true`.

### 5.1.3 Strings

Like numbers and Booleans, strings in GEL can be stored as values inside variables and passed to functions. You can also concatenate a string with another value using the plus operator. For example:

```
a=2+3;"The result is: "+a
```

will create the string:

```
The result is: 5
```

You can also use C-like escape sequences such as `\n`,`\t`,`\b`,`\a` and `\r`. To get a `\` or `"` into the string you can quote it with a `\`. For example:

```
"Slash: \\ Quotes: \" Tabs: \t1\t2\t3"
```

will make a string:

```
Slash: \ Quotes: " Tabs:  1 2 3
```

Do note however that when a string is returned from a function, escapes are quoted, so that the output can be used as input. If you wish to print the string as it is (without escapes), use the print or printn functions.

In addition, you can use the library function string to convert anything to a string. For example:

```
string(22)
```

will return

```
"22"
```

Strings can also be compared with == (equal), != (not equal) and <=> (comparison) operators

### 5.1.4 Null

There is a special value called `null`. No operations can be performed on it, and nothing is printed when it is returned. Therefore, `null` is useful when you do not want output from an expression. The value `null` can be obtained as an expression when you type `.`, the constant `null` or nothing. By nothing we mean that if you end an expression with a separator `;`, it is equivalent to ending it with a separator followed by a `null`.

Example:

```
x=5;.
x=5;
```

Some functions return `null` when no value can be returned or an error happened. Also `null` is used as an empty vector or matrix, or an empty reference.

## 5.2 Using Variables

Syntax:

```
VariableName
```

Example:

```
genius> e
= 2.71828182846
```

To evaluate a variable by itself, just enter the name of the variable. This will return the value of the variable. You can use a variable anywhere you would normally use a number or string. In addition, variables are necessary when defining functions that take arguments (see Section 5.3.1).

---

**Using Tab completion**
You can use Tab completion to get Genius to complete variable names for you. Try typing the first few letters of the name and pressing **`Tab`**.

---

**Variable names are case sensitive**
The names of variables are case sensitive. That means that variables named `hello`, `HELLO` and `Hello` are all different variables.

### 5.2.1 Setting Variables

Syntax:

```
<identifier> = <value>
<identifier> := <value>
```

Example:

```
x = 3
x := 3
```

To assign a value to a variable, use the = or `:=` operators. These operators set the value of the variable and return the value you set, so you can do things like

```
a = b = 5
```

This will set `b` to 5 and then also set `a` to 5.

The = and `:=` operators can both be used to set variables. The difference between them is that the `:=` operator always acts as an assignment operator, whereas the = operator may be interpreted as testing for equality when used in a context where a Boolean expression is expected.

For more information about the scope of variables, that is when are what variables visible, see Section 6.5.

### 5.2.2 Built-in Variables

GEL has a number of built-in 'variables', such as `e`, `pi` or `GoldenRatio`. These are widely used constants with a preset value, and they cannot be assigned new values. There are a number of other built-in variables. See Section 11.4 for a full list. Note that `i` is not by default the square root of negative one (the imaginary number), and is undefined to allow its use as a counter. If you wish to write the imaginary number you need to use **`1i`**.

### 5.2.3 Previous Result Variable

The `Ans` and `ans` variables can be used to get the result of the last expression. For example, if you had performed some calculation, to add 389 to the result you could do:

```
Ans+389
```

## 5.3 Using Functions

Syntax:

```
FunctionName(argument1, argument2, ...)
```

Example:

```
Factorial(5)
cos(2*pi)
gcd(921,317)
```

To evaluate a function, enter the name of the function, followed by the arguments (if any) to the function in parentheses. This will return the result of applying the function to its arguments. The number of arguments to the function is, of course, different for each function.

There are many built-in functions, such as sin, cos and tan. You can use the help built-in command to get a list of available functions, or see Chapter 11 for a full listing.

**Using Tab completion**
You can use Tab completion to get Genius to complete function names for you. Try typing the first few letters of the name and pressing **`Tab`**.

**Function names are case sensitive**
The names of functions are case sensitive. That means that functions named `dosomething`, `DOSOMETHING` and `DoSomething` are all different functions.

### 5.3.1 Defining Functions

Syntax:

```
function <identifier>(<comma separated arguments>) = <function body>
<identifier> = (`() = <function body>)
```

The ` is the backquote character, and signifies an anonymous function. By setting it to a variable name you effectively define a function.

A function takes zero or more comma separated arguments, and returns the result of the function body. Defining your own functions is primarily a matter of convenience; one possible use is to have sets of functions defined in GEL files that Genius can load in order to make them available. Example:

```
function addup(a,b,c) = a+b+c
```

then **`addup(1,4,9)`** yields 14

### 5.3.2 Variable Argument Lists

If you include `...` after the last argument name in the function declaration, then Genius will allow any number of arguments to be passed in place of that argument. If no arguments were passed then that argument will be set to `null`. Otherwise, it will be a horizontal vector containing all the arguments. For example:

```
function f(a,b...) = b
```

Then **`f(1,2,3)`** yields `[2,3]`, while **`f(1)`** yields a `null`.

### 5.3.3 Passing Functions to Functions

In Genius, it is possible to pass a function as an argument to another function. This can be done using either 'function nodes' or anonymous functions.

If you do not enter the parentheses after a function name, instead of being evaluated, the function will instead be returned as a 'function node'. The function node can then be passed to another function. Example:

```
function f(a,b) = a(b)+1;
function b(x) = x*x;
f(b,2)
```

To pass functions that are not defined, you can use an anonymous function (see Section 5.3.1). That is, you want to pass a function without giving it a name. Syntax:

```
function(<comma separated arguments>) = <function body>
`(<comma separated arguments>) = <function body>
```

Example:

```
function f(a,b) = a(b)+1;
f(`(x) = x*x,2)
```

This will return 5.

### 5.3.4 Operations on Functions

Some functions allow arithmetic operations, and some single argument functions such as exp or ln, to operate on the function. For example,

```
exp(sin*cos+4)
```

will return a function that takes `x` and returns **`exp(sin(x)*cos(x)+4)`**. It is functionally equivalent to typing

```
`(x) = exp(sin(x)*cos(x)+4)
```

This operation can be useful when quickly defining functions. For example to create a function called `f` to perform the above operation, you can just type:

```
f = exp(sin*cos+4)
```

It can also be used in plotting. For example, to plot sin squared you can enter:

```
LinePlot(sin^2)
```

**Warning**
Not all functions can be used in this way. For example, when you use a binary operation the functions must take the same number of arguments.

## 5.4 Separator

GEL is somewhat different from other languages in how it deals with multiple commands and functions. In GEL you must chain commands together with a separator operator. That is, if you want to type more than one expression you have to use the **`;`** operator in between the expressions. This is a way in which both expressions are evaluated and the result of the second one (or the last one if there is more than two expressions) is returned. Suppose you type the following:

```
3 ; 5
```

This expression will yield 5.

This will require some parenthesizing to make it unambiguous sometimes, especially if the **`;`** is not the top most primitive. This slightly differs from other programming languages where the **`;`** is a terminator of statements, whereas in GEL it's actually a binary operator. If you are familiar with pascal this should be second nature. However genius can let you pretend it is a terminator to some degree. If a **`;`** is found at the end of a parenthesis or a block, genius will append a null to it as if you would have written **`;null`**. This is useful in case you do not want to return a value from say a loop, or if you handle the return differently. Note that it will slightly slow down the code if it is executed too often as there is one more operator involved.

If you are typing expressions in a program you do not have to add a semicolon. In this case genius will simply print the return value whenever it executes the expression. However, do note that if you are defining a function, the body of the function is a single expression.

## 5.5 Comments

GEL is similar to other scripting languages in that # denotes a comments, that is text that is not meant to be evaluated. Everything beyond the pound sign till the end of line will just be ignored. For example,

```
# This is just a comment
# every line in a comment must have its own pond sign
# in the next line we set x to the value 123
x=123;
```

## 5.6 Modular Evaluation

Genius implements modular arithmetic. To use it you just add "mod <integer>" after the expression. Example: **`2^(5!) * 3^(6!) mod 5`** It could be possible to do modular arithmetic by computing with integers and then modding in the end with the `%` operator, which simply gives the remainder, but that may be time consuming if not impossible when working with larger numbers. For example, **`10^(10^10) % 6`** will simply not work (the exponent will be too large), while **`10^(10^10) mod 6`** is instantaneous. The first expression first tries to compute the integer **`10^(10^10)`** and then find remainder after division by 6, while the second expression evaluates everything modulo 6 to begin with.

You can calculate the inverses of numbers mod some integer by just using rational numbers (of course the inverse has to exist). Examples:

```
10^-1 mod 101
1/10 mod 101
```

You can also do modular evaluation with matrices including taking inverses, powers and dividing. Example:

```
A = [1,2;3,4]
B = A^-1 mod 5
A*B mod 5
```

This should yield the identity matrix as B will be the inverse of A mod 5.

Some functions such as sqrt or log work in a different way when in modulo mode. These will then work like their discrete versions working within the ring of integers you selected. For example:

```
genius> sqrt(4) mod 7
=
[2, 5]
genius> 2*2 mod 7
= 4
```

`sqrt` will actually return all the possible square roots.

Do not chain mod operators, simply place it at the end of the computation, all computations in the expression on the left will be carried out in mod arithmetic. If you place a mod inside a mod, you will get unexpected results. If you simply want to mod a single number and control exactly when remainders are taken, best to use the `%` operator. When you need to chain several expressions in modular arithmetic with different divisors, it may be best to just split up the expression into several and use temporary variables to avoid a mod inside a mod.

## 5.7 List of GEL Operators

Everything in gel is really just an expression. Expressions are stringed together with different operators. As we have seen, even the separator is simply a binary operator in GEL. Here is a list of the operators in GEL.

**`a;b`** The separator, just evaluates both `a` and `b`, but returns only the result of `b`.

**a=b** The assignment operator. This assigns b to a (a must be a valid lvalue) (note however that this operator may be translated to == if used in a place where boolean expression is expected)

**a:=b** The assignment operator. Assigns b to a (a must be a valid lvalue). This is different from = because it never gets translated to a ==.

**|a|** Absolute value. In case the expression is a complex number the result will be the modulus (distance from the origin). For example: **|3 * e^(1i*pi)|** returns 3.

See Mathworld for more information.

**a^b** Exponentiation, raises a to the bth power.

**a.^b** Element by element exponentiation. Raise each element of a matrix a to the bth power. Or if b is a matrix of the same size as a, then do the operation element by element. If a is a number and b is a matrix then it creates matrix of the same size as b with a raised to all the different powers in b.

**a+b** Addition. Adds two numbers, matrices, functions or strings. If you add a string to anything the result will just be a string. If one is a square matrix and the other a number, then the number is multiplied by the identity matrix.

**a-b** Subtraction. Subtract two numbers, matrices or functions.

**a*b** Multiplication. This is the normal matrix multiplication.

**a.*b** Element by element multiplication if a and b are matrices.

**a/b** Division. When a and b are just numbers this is the normal division. When they are matrices, then this is equivalent to **a*b^-1**.

**a./b** Element by element division. Same as **a/b** for numbers, but operates element by element on matrices.

**a\b** Back division. That is this is the same as **b/a**.

**a.\b** Element by element back division.

**a%b** The mod operator. This does not turn on the modular mode, but just returns the remainder of **a/b**.

**a.%b** Element by element the mod operator. Returns the remainder after element by element integer **a./b**.

**a mod b** Modular evaluation operator. The expression a is evaluated modulo b. See Section 5.6. Some functions and operators behave differently modulo an integer.

**a!** Factorial operator. This is like **1*...*(n-2)*(n-1)*n**.

**a!!** Double factorial operator. This is like **1*...*(n-4)*(n-2)*n**.

**a==b** Equality operator. Returns true or false depending on a and b being equal or not.

**a!=b** Inequality operator, returns true if a does not equal b else returns false.

**a<>b** Alternative inequality operator, returns true if a does not equal b else returns false.

**a<=b** Less than or equal operator, returns true if a is less than or equal to b else returns false. These can be chained as in **a <= b <= c** (can also be combined with the less than operator).

**a>=b** Greater than or equal operator, returns true if a is greater than or equal to b else returns false. These can be chained as in **a >= b >= c** (can also be combine with the greater than operator).

**a<b** Less than operator, returns true if a is less than b else returns false. These can be chained as in **a < b < c** (can also be combine with the less than or equal to operator).

**a>b** Greater than operator, returns true if a is greater than b else returns false. These can be chained as in **a > b > c** (can also be combine with the greater than or equal to operator).

**a<=>b** Comparison operator. If a is equal to b it returns 0, if a is less than b it returns -1 and if a is greater than b it returns 1.

**a and b** Logical and. Returns true if both a and b are true, else returns false. If given numbers, nonzero numbers are treated as true.

**a or b** Logical or. Returns true if both a or b are true, else returns false. If given numbers, nonzero numbers are treated as true.

**a xor b** Logical xor. Returns true exactly one of a or b is true, else returns false. If given numbers, nonzero numbers are treated as true.

**not a** Logical not. Returns the logical negation of a

**-a** Negation operator. Returns the negative of a number or a matrix (works element-wise on a matrix).

**&a** Variable referencing (to pass a reference to a variable). See Section 6.8.

***a** Variable dereferencing (to access a referenced variable). See Section 6.8.

**a'** Matrix conjugate transpose. That is, rows and columns get swapped and we take complex conjugate of all entries. That is if the i,j element of a is x+iy, then the j,i element of **a'** is x-iy.

**a.'** Matrix transpose, does not conjugate the entries. That is, the i,j element of a becomes the j,i element of **a.'**.

**a@(b,c)** Get element of a matrix in row b and column c. If b, c are vectors, then this gets the corresponding rows columns or submatrices.

**a@(b,)** Get row of a matrix (or multiple rows if b is a vector).

**a@(b,:)** Same as above.

**a@(,c)** Get column of a matrix (or columns if c is a vector).

**a@(:,c)** Same as above.

**a@(b)** Get an element from a matrix treating it as a vector. This will traverse the matrix row-wise.

**a:b** Build a vector from a to b (or specify a row, column region for the @ operator). For example to get rows 2 to 4 of matrix A we could do

```
A@(2:4,)
```

as **2:4** will return a vector **[2,3,4]**.

**a:b:c** Build a vector from a to c with b as a step. That is for example

```
genius> 1:2:9
=
`[1, 3, 5, 7, 9]
```

When the numbers involved are floating point numbers, for example **1.0:0.4:3.0**, the output is what is expected even though adding 0.4 to 1.0 five times is actually just slightly more than 3.0 due to the way that floating point numbers are stored in base 2 (there is no 0.4, the actual number stored is just ever so slightly bigger). The way this is handled is the same as in the for, sum, and prod loops. If the end is within **2^-20** times the step size of the endpoint, the endpoint is used and we assume there were roundoff errors. This is not perfect, but it handles the majority of the cases. This check is done only from version 1.0.18 onwards, so execution of your code may differ on older versions. If you want to avoid dealing with this issue, use actual rational numbers, possibly using the float if you wish to get floating point numbers in the end. For example **1:2/5:3** does the right thing and **float(1:2/5:3)** even gives you floating point numbers and is ever so slightly more precise than **1.0:0.4:3.0**.

**(a)i** Make a imaginary number (multiply a by the imaginary). Note that normally the number i is written as **1i**. So the above is equal to

```
(a)*1i
```

**`a** Quote an identifier so that it doesn't get evaluated. Or quote a matrix so that it doesn't get expanded.

**a swapwith b** Swap value of a with the value of b. Currently does not operate on ranges of matrix elements. It returns null. Available from version 1.0.13.

**increment a** Increment the variable a by 1. If a is a matrix, then increment each element. This is equivalent to **a=a+1**, but it is somewhat faster. It returns null. Available from version 1.0.13.

**increment a by b** Increment the variable a by b. If a is a matrix, then increment each element. This is equivalent to **a=a+b**, but it is somewhat faster. It returns null. Available from version 1.0.13.

---

**Note**

The @() operator makes the : operator most useful. With this you can specify regions of a matrix. So that a@(2:4,6) is the rows 2,3,4 of the column 6. Or a@(,1:2) will get you the first two columns of a matrix. You can also assign to the @() operator, as long as the right value is a matrix that matches the region in size, or if it is any other type of value.

---

**Note**

The comparison operators (except for the <=> operator, which behaves normally), are not strictly binary operators, they can in fact be grouped in the normal mathematical way, e.g.: (1<x<=y<5) is a legal boolean expression and means just what it should, that is (1<x and x$\leq$y and y<5)

---

**Note**

The unitary minus operates in a different fashion depending on where it appears. If it appears before a number it binds very closely, if it appears in front of an expression it binds less than the power and factorial operators. So for example **-1^k** is really **(-1)^k**, but **-foo(1)^k** is really **-(foo(1)^k)**. So be careful how you use it and if in doubt, add parentheses.

---

# Chapter 6

# Programming with GEL

## 6.1 Conditionals

Syntax:

```
if <expression1> then <expression2> [else <expression3>]
```

If `else` is omitted, then if the `expression1` yields `false` or 0, `NULL` is returned.

Examples:

```
if(a==5)then(a=a-1)
if b<a then b=a
if c>0 then c=c-1 else c=0
a = ( if b>0 then b else 1 )
```

Note that = will be translated to == if used inside the expression for `if`, so

```
if a=5 then a=a-1
```

will be interpreted as:

```
if a==5 then a:=a-1
```

## 6.2 Loops

### 6.2.1 While Loops

Syntax:

```
while <expression1> do <expression2>
until <expression1> do <expression2>
do <expression2> while <expression1>
do <expression2> until <expression1>
```

These are similar to other languages. However, as in gel it is simply an expression that must have some return value, these constructs will simply return the result of the last iteration or `NULL` if no iteration was done. In the boolean expression, = is translated into == just as for the `if` statement.

### 6.2.2 For Loops

Syntax:

```
for <identifier> = <from> to <to> do <body>
for <identifier> = <from> to <to> by <increment> do <body>
```

Loop with identifier being set to all values from `<from>` to `<to>`, optionally using an increment other than 1. These are faster, nicer and more compact than the normal loops such as above, but less flexible. The identifier must be an identifier and can't be a dereference. The value of identifier is the last value of identifier, or `<from>` if body was never evaluated. The variable is guaranteed to be initialized after a loop, so you can safely use it. Also the `<from>`, `<to>` and `<increment>` must be non complex values. The `<to>` is not guaranteed to be hit, but will never be overshot, for example the following prints out odd numbers from 1 to 19:

```
for i = 1 to 20 by 2 do print(i)
```

When one of the values is a floating point number, then the final check is done to within 2^-20 of the step size. That is, even if we overshoot by 2^-20 times the "by" above, we still execute the last iteration. This way

```
for x = 0 to 1 by 0.1 do print(x)
```

does the expected even though adding 0.1 ten times becomes just slightly more 1.0 due to the way that floating point numbers are stored in base 2 (there is no 0.1, the actual number stored is just ever so slightly bigger). This is not perfect but it handles the majority of the cases. If you want to avoid dealing with this issue, use actual rational numbers for example:

```
for x = 0 to 1 by 1/10 do print(x)
```

This check is done only from version 1.0.16 onwards, so execution of your code may differ on older versions.

### 6.2.3 Foreach Loops

Syntax:

```
for <identifier> in <matrix> do <body>
```

For each element in the matrix, going row by row from left to right we execute the body with the identifier set to the current element. To print numbers 1,2,3 and 4 in this order you could do:

```
for n in [1,2:3,4] do print(n)
```

If you wish to run through the rows and columns of a matrix, you can use the RowsOf and ColumnsOf functions, which return a vector of the rows or columns of the matrix. So,

```
for n in RowsOf ([1,2:3,4]) do print(n)
```

will print out [1,2] and then [3,4].

### 6.2.4 Break and Continue

You can also use the `break` and `continue` commands in loops. The continue `continue` command will restart the current loop at its next iteration, while the `break` command exits the current loop.

```
while(<expression1>) do (
  if(<expression2>) break
  else if(<expression3>) continue;
  <expression4>
)
```

## 6.3 Sums and Products

Syntax:

```
sum <identifier> = <from> to <to> do <body>
sum <identifier> = <from> to <to> by <increment> do <body>
sum <identifier> in <matrix> do <body>
prod <identifier> = <from> to <to> do <body>
prod <identifier> = <from> to <to> by <increment> do <body>
prod <identifier> in <matrix> do <body>
```

If you substitute `for` with `sum` or `prod`, then you will get a sum or a product instead of a `for` loop. Instead of returning the last value, these will return the sum or the product of the values respectively.

If no body is executed (for example **`sum i=1 to 0 do ...`**) then `sum` returns 0 and `prod` returns 1 as is the standard convention.

For floating point numbers the same roundoff error protection is done as in the for loop. See Section 6.2.2.

## 6.4 Comparison Operators

The following standard comparison operators are supported in GEL and have the obvious meaning: ==, >=, <=, !=, <>, <, >. They return `true` or `false`. The operators != and <> are the same thing and mean "is not equal to". GEL also supports the operator <=>, which returns -1 if left side is smaller, 0 if both sides are equal, 1 if left side is larger.

Normally = is translated to == if it happens to be somewhere where GEL is expecting a condition such as in the if condition. For example

```
if a=b then c
if a==b then c
```

are the same thing in GEL. However you should really use == or := when you want to compare or assign respectively if you want your code to be easy to read and to avoid mistakes.

All the comparison operators (except for the <=> operator, which behaves normally), are not strictly binary operators, they can in fact be grouped in the normal mathematical way, e.g.: (1<x<=y<5) is a legal boolean expression and means just what it should, that is (1<x and x≤y and y<5)

To build up logical expressions use the words `not`, `and`, `or`, `xor`. The operators `or` and `and` are special beasts as they evaluate their arguments one by one, so the usual trick for conditional evaluation works here as well. For example, `1 or a=1` will not set `a=1` since the first argument was true.

## 6.5 Global Variables and Scope of Variables

GEL is a dynamically scoped language. We will explain what this means below. That is, normal variables and functions are dynamically scoped. The exception are parameter variables, which are always global.

Like most programming languages, GEL has different types of variables. Normally when a variable is defined in a function, it is visible from that function and from all functions that are called (all higher contexts). For example, suppose a function `f` defines a variable `a` and then calls function `g`. Then function `g` can reference `a`. But once `f` returns, the variable `a` goes out of scope. For example, the following code will print out 5. The function `g` cannot be called on the top level (outside `f` as `a` will not be defined).

```
function f() = (a:=5; g());
function g() = print(a);
f();
```

If you define a variable inside a function it will override any variables defined in calling functions. For example, we modify the above code and write:

```
function f() = (a:=5; g());
function g() = print(a);
a:=10;
f();
```

This code will still print out 5. But if you call `g` outside of `f` then you will get a printout of 10. Note that setting `a` to 5 inside `f` does not change the value of `a` at the top (global) level, so if you now check the value of `a` it will still be 10.

Function arguments are exactly like variables defined inside the function, except that they are initialized with the value that was passed to the function. Other than this point, they are treated just like all other variables defined inside the function.

Functions are treated exactly like variables. Hence you can locally redefine functions. Normally (on the top level) you cannot redefine protected variables and functions. But locally you can do this. Consider the following session:

```
genius> function f(x) = sin(x)^2
= (`(x)=(sin(x)^2))
genius> function f(x) = sin(x)^2
= (`(x)=(sin(x)^2))
genius> function g(x) = ((function sin(x)=x^10);f(x))
= (`(x)=((sin:=(`(x)=(x^10)));f(x)))
genius> g(10)
= 1e20
```

Functions and variables defined at the top level are considered global. They are visible from anywhere. As we said the following function `f` will not change the value of `a` to 5.

```
a=6;
function f() = (a:=5);
f();
```

Sometimes, however, it is necessary to set a global variable from inside a function. When this behavior is needed, use the set function. Passing a string or a quoted identifier to this function sets the variable globally (on the top level). For example, to set `a` to the value 3 you could call:

```
set(`a,3)
```

or:

```
set("a",3)
```

The `set` function always sets the toplevel global. There is no way to set a local variable in some function from a subroutine. If this is required, must use passing by reference.

See also the SetElement and SetVElement functions.

So to recap in a more technical language: Genius operates with different numbered contexts. The top level is the context 0 (zero). Whenever a function is entered, the context is raised, and when the function returns the context is lowered. A function or a variable is always visible from all higher numbered contexts. When a variable was defined in a lower numbered context, then setting this variable has the effect of creating a new local variable in the current context number and this variable will now be visible from all higher numbered contexts.

There are also true local variables that are not seen from anywhere but the current context. Also when returning functions by value it may reference variables not visible from higher context and this may be a problem. See the sections True Local Variables and Returning Functions.

## 6.6 Parameter variables

As we said before, there exist special variables called parameters that exist in all scopes. To declare a parameter called `foo` with the initial value 1, we write

```
parameter foo = 1
```

From then on, `foo` is a strictly global variable. Setting `foo` inside any function will modify the variable in all contexts, that is, functions do not have a private copy of parameters.

When you undefine a parameter using the undefine function, it stops being a parameter.

Some parameters are built-in and modify the behavior of genius.

## 6.7 Returning

Normally a function is one or several expressions separated by a semicolon, and the value of the last expression is returned. This is fine for simple functions, but sometimes you do not want a function to return the last thing calculated. You may, for example, want to return from a middle of a function. In this case, you can use the `return` keyword. `return` takes one argument, which is the value to be returned.

Example:

```
function f(x) = (
  y=1;
  while true do (
    if x>50 then return y;
    y=y+1;
    x=x+1
  )
)
```

## 6.8 References

It may be necessary for some functions to return more than one value. This may be accomplished by returning a vector of values, but many times it is convenient to use passing a reference to a variable. You pass a reference to a variable to a function, and the function will set the variable for you using a dereference. You do not have to use references only for this purpose, but this is their main use.

When using functions that return values through references in the argument list, just pass the variable name with an ampersand. For example the following code will compute an eigenvalue of a matrix `A` with initial eigenvector guess `x`, and store the computed eigenvector into the variable named `v`:

```
RayleighQuotientIteration (A,x,0.001,100,&v)
```

The details of how references work and the syntax is similar to the C language. The operator `&` references a variable and `*` dereferences a variable. Both can only be applied to an identifier, so `**a` is not a legal expression in GEL.

References are best explained by an example:

```
a=1;
b=&a;
*b=2;
```

now `a` contains 2. You can also reference functions:

```
function f(x) = x+1;
t=&f;
*t(3)
```

gives us 4.

## 6.9 Lvalues

An lvalue is the left hand side of an assignment. In other words, an lvalue is what you assign something to. Valid lvalues are:

**a** Identifier. Here we would be setting the variable of name a.

***a** Dereference of an identifier. This will set whatever variable a points to.

**a@(<region>)** A region of a matrix. Here the region is specified normally as with the regular @() operator, and can be a single entry, or an entire region of the matrix.

Examples:

```
a:=4
*tmp := 89
a@(1,1) := 5
a@(4:8,3) := [1,2,3,4,5]'
```

Note that both := and = can be used interchangeably. Except if the assignment appears in a condition. It is thus always safer to just use := when you mean assignment, and == when you mean comparison.

# Chapter 7

# Advanced Programming with GEL

## 7.1 Error Handling

If you detect an error in your function, you can bail out of it. For normal errors, such as wrong types of arguments, you can fail to compute the function by adding the statement `bailout`. If something went really wrong and you want to completely kill the current computation, you can use `exception`.

For example if you want to check for arguments in your function. You could use the following code.

```
function f(M) = (
  if not IsMatrix (M) then (
    error ("M not a matrix!");
    bailout
  );
  ...
)
```

## 7.2 Toplevel Syntax

The syntax is slightly different if you enter statements on the top level versus when they are inside parentheses or inside functions. On the top level, enter acts the same as if you press return on the command line. Therefore think of programs as just sequence of lines as if were entered on the command line. In particular, you do not need to enter the separator at the end of the line (unless it is of course part of several statements inside parentheses).

The following code will produce an error when entered on the top level of a program, while it will work just fine in a function.

```
if Something() then
  DoSomething()
else
  DoSomethingElse()
```

The problem is that after Genius Mathematics Tool sees the end of line after the second line, it will decide that we have whole statement and it will execute it. After the execution is done, Genius Mathematics Tool will go on to the next line, it will see `else`, and it will produce a parsing error. To fix this, use parentheses. Genius Mathematics Tool will not be satisfied until it has found that all parentheses are closed.

```
if Something() then (
  DoSomething()
) else (
  DoSomethingElse()
)
```

## 7.3 Returning Functions

It is possible to return functions as value. This way you can build functions that construct special purpose functions according to some parameters. The tricky bit is what variables does the function see. The way this works in GEL is that when a function returns another function, all identifiers referenced in the function body that went out of scope are prepended a private dictionary of the returned function. So the function will see all variables that were in scope when it was defined. For example, we define a function that returns a function that adds 5 to its argument.

```
function f() = (
  k = 5;
  `(x) = (x+k)
)
```

Notice that the function adds `k` to `x`. You could use this as follows.

```
g = f();
g(5)
```

And **`g(5)`** should return 10.

One thing to note is that the value of `k` that is used is the one that's in effect when the `f` returns. For example:

```
function f() = (
  k := 5;
  function r(x) = (x+k);
  k := 10;
  r
)
```

will return a function that adds 10 to its argument rather than 5. This is because the extra dictionary is created only when the context in which the function was defined ends, which is when the function `f` returns. This is consistent with how you would expect the function `r` to work inside the function `f` according to the rules of scope of variables in GEL. Only those variables are added to the extra dictionary that are in the context that just ended and no longer exists. Variables used in the function that are in still valid contexts will work as usual, using the current value of the variable. The only difference is with global variables and functions. All identifiers that referenced global variables at time of the function definition are not added to the private dictionary. This is to avoid much unnecessary work when returning functions and would rarely be a problem. For example, suppose that you delete the "k=5" from the function `f`, and at the top level you define `k` to be say 5. Then when you run `f`, the function `r` will not put `k` into the private dictionary because it was global (toplevel) at the time of definition of `r`.

Sometimes it is better to have more control over how variables are copied into the private dictionary. Since version 1.0.7, you can specify which variables are copied into the private dictionary by putting extra square brackets after the arguments with the list of variables to be copied separated by commas. If you do this, then variables are copied into the private dictionary at time of the function definition, and the private dictionary is not touched afterwards. For example

```
function f() = (
  k := 5;
  function r(x) [k] = (x+k);
  k := 10;
  r
)
```

will return a function that when called will add 5 to its argument. The local copy of `k` was created when the function was defined.

When you want the function to not have any private dictionary then put empty square brackets after the argument list. Then no private dictionary will be created at all. Doing this is good to increase efficiency when a private dictionary is not needed or when you want the function to lookup all variables as it sees them when called. For example suppose you want the function returned from `f` to see the value of `k` from the toplevel despite there being a local variable of the same name during definition. So the code

```
function f() = (
  k := 5;
  function r(x) [] = (x+k);
```

```
  r
);
k := 10;
g = f();
g(10)
```

will return 20 and not 15, which would happen if `k` with a value of 5 was added to the private dictionary.

## 7.4 True Local Variables

When passing functions into other functions, the normal scoping of variables might be undesired. For example:

```
k := 10;
function r(x) = (x+k);
function f(g,x) = (
  k := 5;
  g(x)
);
f(r,1)
```

you probably want the function `r` when passed as `g` into `f` to see `k` as 10 rather than 5, so that the code returns 11 and not 6. However, as written, the function when executed will see the `k` that is equal to 5. There are two ways to solve this. One would be to have `r` get `k` in a private dictionary using the square bracket notation section Returning Functions.

But there is another solution. Since version 1.0.7 there are true local variables. These are variables that are visible only from the current context and not from any called functions. We could define `k` as a local variable in the function `f`. To do this add a **local** statement as the first statement in the function (it must always be the first statement in the function). You can also make any arguments be local variables as well. That is,

```
function f(g,x) = (
  local g,x,k;
  k := 5;
  g(x)
);
```

Then the code will work as expected and prints out 11. Note that the **local** statement initializes all the referenced variables (except for function arguments) to a `null`.

If all variables are to be created as locals you can just pass an asterisk instead of a list of variables. In this case the variables will not be initialized until they are actually set of course. So the following definition of `f` will also work:

```
function f(g,x) = (
  local *;
  k := 5;
  g(x)
);
```

It is good practice that all functions that take other functions as arguments use local variables. This way the passed function does not see implementation details and get confused.

## 7.5 GEL Startup Procedure

First the program looks for the installed library file (the compiled version `lib.cgel`) in the installed directory, then it looks into the current directory, and then it tries to load an uncompiled file called `~/.geniusinit`.

If you ever change the library in its installed place, you'll have to first compile it with **genius --compile loader.gel > lib.cgel**

## 7.6 Loading Programs

Sometimes you have a larger program you wrote into a file and want to read that file into Genius Mathematics Tool. In these situations, you have two options. You can keep the functions you use most inside the `~/.geniusinit` file. Or if you want to load up a file in a middle of a session (or from within another file), you can type **load <list of filenames>** at the prompt. This has to be done on the top level and not inside any function or whatnot, and it cannot be part of any expression. It also has a slightly different syntax than the rest of genius, more similar to a shell. You can enter the file in quotes. If you use the ” quotes, you will get exactly the string that you typed, if you use the "" quotes, special characters will be unescaped as they are for strings. Example:

```
load program1.gel program2.gel
load "Weird File Name With SPACES.gel"
```

There are also **cd**, **pwd** and **ls** commands built in. **cd** will take one argument, **ls** will take an argument that is like the glob in the UNIX shell (i.e., you can use wildcards). **pwd** takes no arguments. For example:

```
cd directory_with_gel_programs
ls *.gel
```

# Chapter 8

# Matrices in GEL

Genius has support for vectors and matrices and possesses a sizable library of matrix manipulation and linear algebra functions.

## 8.1 Entering Matrices

To enter matrices, you can use one of the following two syntaxes. You can either enter the matrix on one line, separating values by commas and rows by semicolons. Or you can enter each row on one line, separating values by commas. You can also just combine the two methods. So to enter a 3x3 matrix of numbers 1-9 you could do

```
[1,2,3;4,5,6;7,8,9]
```

or

```
[1, 2, 3
 4, 5, 6
 7, 8, 9]
```

Do not use both ';' and return at once on the same line though.

You can also use the matrix expansion functionality to enter matrices. For example you can do:

```
a = [ 1, 2, 3
      4, 5, 6
      7, 8, 9]
b = [ a,  10
      11, 12]
```

and you should get

```
[1,   2,  3, 10
 4,   5,  6, 10
 7,   8,  9, 10
 11, 11, 11, 12]
```

similarly you can build matrices out of vectors and other stuff like that.

Another thing is that non-specified spots are initialized to 0, so

```
[1, 2, 3
 4, 5
 6]
```

will end up being

```
[1, 2, 3
 4, 5, 0
 6, 0, 0]
```

When matrices are evaluated, they are evaluated and traversed row-wise. This is just like the `M@(j)` operator, which traverses the matrix row-wise.

---

**Note**
Be careful about using returns for expressions inside the `[ ]` brackets, as they have a slightly different meaning there. You will start a new row.

---

## 8.2 Conjugate Transpose and Transpose Operator

You can conjugate transpose a matrix by using the `'` operator. That is the entry in the `i`th column and the `j`th row will be the complex conjugate of the entry in the `j`th column and the `i`th row of the original matrix. For example:

```
[1,2,3]*[4,5,6]'
```

We transpose the second vector to make matrix multiplication possible. If you just want to transpose a matrix without conjugating it, you would use the `.'` operator. For example:

```
[1,2,3]*[4,5,6i].'
```

Note that normal transpose, that is the `.'` operator, is much faster and will not create a new copy of the matrix in memory. The conjugate transpose does create a new copy unfortunately. It is recommended to always use the `.'` operator when working with real matrices and vectors.

## 8.3 Linear Algebra

Genius implements many useful linear algebra and matrix manipulation routines. See the Linear Algebra and Matrix Manipulation sections of the GEL function listing.

The linear algebra routines implemented in GEL do not currently come from a well tested numerical package, and thus should not be used for critical numerical computation. On the other hand, Genius implements very well many linear algebra operations with rational and integer coefficients. These are inherently exact and in fact will give you much better results than common double precision routines for linear algebra.

For example, it is pointless to compute the rank and nullspace of a floating point matrix since for all practical purposes, we need to consider the matrix as having some slight errors. You are likely to get a different result than you expect. The problem is that under a small perturbation every matrix is of full rank and invertible. If the matrix however is of rational numbers, then the rank and nullspace are always exact.

In general when Genius computes the basis of a certain vectorspace (for example with the NullSpace) it will give the basis as a matrix, in which the columns are the vectors of the basis. That is, when Genius talks of a linear subspace it means a matrix whose column space is the given linear subspace.

It should be noted that Genius can remember certain properties of a matrix. For example, it will remember that a matrix is in row reduced form. If many calls are made to functions that internally use row reduced form of the matrix, we can just row reduce the matrix beforehand once. Successive calls to rref will be very fast.

# Chapter 9

# Polynomials in GEL

Currently Genius can handle polynomials of one variable written out as vectors, and do some basic operations with these. It is planned to expand this support further.

## 9.1 Using Polynomials

Currently polynomials in one variable are just horizontal vectors with value only nodes. The power of the term is the position in the vector, with the first position being 0. So,

```
[1,2,3]
```

translates to a polynomial of

```
1 + 2*x + 3*x^2
```

You can add, subtract and multiply polynomials using the AddPoly, SubtractPoly, and MultiplyPoly functions respectively. You can print a polynomial using the PolyToString function. For example,

```
PolyToString([1,2,3],"y")
```

gives

```
3*y^2 + 2*y + 1
```

You can also get a function representation of the polynomial so that you can evaluate it. This is done by using PolyToFunction, which returns an anonymous function.

```
f = PolyToFunction([0,1,1])
f(2)
```

It is also possible to find roots of polynomials of degrees 1 through 4 by using the function PolynomialRoots, which calls the appropriate formula function. Higher degree polynomials must be converted to functions and solved numerically using a function such as FindRootBisection, FindRootFalsePosition, FindRootMullersMethod, or FindRootSecant.

See Section 11.15 in the function list for the rest of functions acting on polynomials.

# Chapter 10

# Set Theory in GEL

Genius has some basic set theoretic functionality built in. Currently a set is just a vector (or a matrix). Every distinct object is treated as a different element.

## 10.1 Using Sets

Just like vectors, objects in sets can include numbers, strings, `null`, matrices and vectors. It is planned in the future to have a dedicated type for sets, rather than using vectors. Note that floating point numbers are distinct from integers, even if they appear the same. That is, Genius will treat `0` and `0.0` as two distinct elements. The `null` is treated as an empty set.

To build a set out of a vector, use the MakeSet function. Currently, it will just return a new vector where every element is unique.

```
genius> MakeSet([1,2,2,3])
= [1, 2, 3]
```

Similarly there are functions Union, Intersection, SetMinus, which are rather self explanatory. For example:

```
genius> Union([1,2,3], [1,2,4])
= [1, 2, 4, 3]
```

Note that no order is guaranteed for the return values. If you wish to sort the vector you should use the SortVector function.

For testing membership, there are functions IsIn and IsSubset, which return a boolean value. For example:

```
genius> IsIn (1, [0,1,2])
= true
```

The input **`IsIn(x,X)`** is of course equivalent to **`IsSubset([x],X)`**. Note that since the empty set is a subset of every set, **`IsSubset(null,X)`** is always true.

# Chapter 11

# List of GEL functions

To get help on a specific function from the console type:

```
help FunctionName
```

## 11.1 Commands

**help** `help`

```
help FunctionName
```

Print help (or help on a function/command).

**load** `load "file.gel"`

Load a file into the interpreter. The file will execute as if it were typed onto the command line.

**cd** `cd /directory/name`

Change working directory to `/directory/name`.

**pwd** `pwd`

Print the current working directory.

**ls** `ls`

List files in the current directory.

**plugin** `plugin plugin_name`

Load a plugin. Plugin of that name must be installed on the system in the proper directory.

## 11.2 Basic

**AskButtons** `AskButtons (query)`

```
AskButtons (query, button1, ...)
```

Asks a question and presents a list of buttons to the user (or a menu of options in text mode). Returns the 1-based index of the button pressed. That is, returns 1 if the first button was pressed, 2 if the second button was pressed, and so on. If the user closes the window (or simply hits enter in text mode), then `null` is returned. The execution of the program is blocked until the user responds.

Version 1.0.10 onwards.

**AskString** `AskString (query)`

`AskString (query, default)`

Asks a question and lets the user enter a string, which it then returns. If the user cancels or closes the window, then `null` is returned. The execution of the program is blocked until the user responds. If `default` is given, then it is pre-typed in for the user to just press enter on (version 1.0.6 onwards).

**Compose** `Compose (f,g)`

Compose two functions and return a function that is the composition of `f` and `g`.

**ComposePower** `ComposePower (f,n,x)`

Compose and execute a function with itself `n` times, passing `x` as argument. Returning `x` if `n` equals 0. Example:

```
genius> function f(x) = x^2 ;
genius> ComposePower (f,3,7)
= 5764801
genius> f(f(f(7)))
= 5764801
```

**Evaluate** `Evaluate (str)`

Parses and evaluates a string.

**GetCurrentModulo** `GetCurrentModulo`

Get current modulo from the context outside the function. That is, if outside of the function was executed in modulo (using `mod`) then this returns what this modulo was. Normally the body of the function called is not executed in modular arithmetic, and this builtin function makes it possible to make GEL functions aware of modular arithmetic.

**Identity** `Identity (x)`

Identity function, returns its argument. It is equivalent to **`function Identity(x)=x`**.

**IntegerFromBoolean** `IntegerFromBoolean (bval)`

Make integer (0 for `false` or 1 for `true`) from a boolean value. `bval` can also be a number in which case a non-zero value will be interpreted as `true` and zero will be interpreted as `false`.

**IsBoolean** `IsBoolean (arg)`

Check if argument is a boolean (and not a number).

**IsDefined** `IsDefined (id)`

Check if an id is defined. You should pass a string or and identifier. If you pass a matrix, each entry will be evaluated separately and the matrix should contain strings or identifiers.

**IsFunction** `IsFunction (arg)`

Check if argument is a function.

**IsFunctionOrIdentifier** `IsFunctionOrIdentifier (arg)`

Check if argument is a function or an identifier.

**IsFunctionRef** `IsFunctionRef (arg)`

Check if argument is a function reference. This includes variable references.

**IsMatrix** `IsMatrix (arg)`

Check if argument is a matrix. Even though `null` is sometimes considered an empty matrix, the function `IsMatrix` does not consider `null` a matrix.

**IsNull** `IsNull (arg)`

Check if argument is a `null`.

**IsString** `IsString (arg)`

Check if argument is a text string.

**IsValue** `IsValue (arg)`

Check if argument is a number.

**Parse** `Parse (str)`

Parses but does not evaluate a string. Note that certain pre-computation is done during the parsing stage.

**SetFunctionFlags** `SetFunctionFlags (id,flags...)`

Set flags for a function, currently `"PropagateMod"` and `"NoModuloArguments"`. If `"PropagateMod"` is set, then the body of the function is evaluated in modular arithmetic when the function is called inside a block that was evaluated using modular arithmetic (using `mod`). If `"NoModuloArguments"`, then the arguments of the function are never evaluated using modular arithmetic.

**SetHelp** `SetHelp (id,category,desc)`

Set the category and help description line for a function.

**SetHelpAlias** `SetHelpAlias (id,alias)`

Sets up a help alias.

**chdir** `chdir (dir)`

Changes current directory, same as the **cd**.

**CurrentTime** `CurrentTime`

Returns the current UNIX time with microsecond precision as a floating point number. That is, returns the number of seconds since January 1st 1970.

Version 1.0.15 onwards.

**display** `display (str,expr)`

Display a string and an expression with a colon to separate them.

**DisplayVariables** `DisplayVariables (var1,var2,...)`

Display set of variables. The variables can be given as strings or identifiers. For example:

```
DisplayVariables(`x,`y,`z)
```

If called without arguments (must supply empty argument list) as

```
DisplayVariables()
```

then all variables are printed including a stacktrace similar to Show user variables in the graphical version.

Version 1.0.18 onwards.

**error** `error (str)`

Prints a string to the error stream (onto the console).

**exit** `exit`

Aliases: `quit`

Exits the program.

**false** `false`

Aliases: `False` `FALSE`

The `false` boolean value.

**manual** `manual`

Displays the user manual.

**print** `print (str)`

Prints an expression and then print a newline. The argument `str` can be any expression. It is made into a string before being printed.

**printn** `printn (str)`

Prints an expression without a trailing newline. The argument `str` can be any expression. It is made into a string before being printed.

**PrintTable** `PrintTable (f,v)`

Print a table of values for a function. The values are in the vector `v`. You can use the vector building notation as follows:

```
PrintTable (f,[0:10])
```

If `v` is a positive integer, then the table of integers from 1 up to and including v will be used.

Version 1.0.18 onwards.

**protect** `protect (id)`

Protect a variable from being modified. This is used on the internal GEL functions to avoid them being accidentally overridden.

**ProtectAll** `ProtectAll ()`

Protect all currently defined variables, parameters and functions from being modified. This is used on the internal GEL functions to avoid them being accidentally overridden. Normally Genius Mathematics Tool considers unprotected variables as user defined.

Version 1.0.7 onwards.

**set** `set (id,val)`

Set a global variable. The `id` can be either a string or a quoted identifier. For example:

```
set('x,1)
```

will set the global variable `x` to the value 1.

The function returns the `val`, to be usable in chaining.

**SetElement** `SetElement (id,row,col,val)`

Set an element of a global variable which is a matrix. The `id` can be either a string or a quoted identifier. For example:

```
SetElement('x,2,3,1)
```

will set the second row third column element of the global variable `x` to the value 1. If no global variable of the name exists, or if it is set to something that's not a matrix, a new zero matrix of appropriate size will be created.

The `row` and `col` can also be ranges, and the semantics are the same as for regular setting of the elements with an equals sign.

The function returns the `val`, to be usable in chaining.

Available from 1.0.18 onwards.

**SetVElement** `SetElement (id,elt,val)`

Set an element of a global variable which is a vector. The `id` can be either a string or a quoted identifier. For example:

```
SetElement(`x,2,1)
```

will set the second element of the global vector variable `x` to the value 1. If no global variable of the name exists, or if it is set to something that's not a vector (matrix), a new zero row vector of appropriate size will be created.

The `elt` can also be a range, and the semantics are the same as for regular setting of the elements with an equals sign.

The function returns the `val`, to be usable in chaining.

Available from 1.0.18 onwards.

**string** `string (s)`

Make a string. This will make a string out of any argument.

**true** `true`

Aliases: `True` `TRUE`

The `true` boolean value.

**undefine** `undefine (id)`

Alias: `Undefine`

Undefine a variable. This includes locals and globals, every value on all context levels is wiped. This function should really not be used on local variables. A vector of identifiers can also be passed to undefine several variables.

**UndefineAll** `UndefineAll ()`

Undefine all unprotected global variables (including functions and parameters). Normally Genius Mathematics Tool considers protected variables as system defined functions and variables. Note that `UndefineAll` only removes the global definition of symbols not local ones, so that it may be run from inside other functions safely.

Version 1.0.7 onwards.

**unprotect** `unprotect (id)`

Unprotect a variable from being modified.

**UserVariables** `UserVariables ()`

Return a vector of identifiers of user defined (unprotected) global variables.

Version 1.0.7 onwards.

**wait** `wait (secs)`

Waits a specified number of seconds. `secs` must be non-negative. Zero is accepted and nothing happens in this case, except possibly user interface events are processed.

Since version 1.0.18, `secs` can be a noninteger number, so **`wait(0.1)`** will wait for one tenth of a second.

**version** `version`

Returns the version of Genius as a horizontal 3-vector with major version first, then minor version and finally the patch level.

**warranty** `warranty`

Gives the warranty information.

## 11.3 Parameters

**ChopTolerance** `ChopTolerance = number`

Tolerance of the `Chop` function.

**ContinuousNumberOfTries** `ContinuousNumberOfTries = number`

How many iterations to try to find the limit for continuity and limits.

**ContinuousSFS** `ContinuousSFS = number`

How many successive steps to be within tolerance for calculation of continuity.

**ContinuousTolerance** `ContinuousTolerance = number`

Tolerance for continuity of functions and for calculating the limit.

**DerivativeNumberOfTries** `DerivativeNumberOfTries = number`

How many iterations to try to find the limit for derivative.

**DerivativeSFS** `DerivativeSFS = number`

How many successive steps to be within tolerance for calculation of derivative.

**DerivativeTolerance** `DerivativeTolerance = number`

Tolerance for calculating the derivatives of functions.

**ErrorFunctionTolerance** `ErrorFunctionTolerance = number`

Tolerance of the ErrorFunction.

**FloatPrecision** `FloatPrecision = number`

Floating point precision.

**FullExpressions** `FullExpressions = boolean`

Print full expressions, even if more than a line.

**GaussDistributionTolerance** `GaussDistributionTolerance = number`

Tolerance of the GaussDistribution function.

**IntegerOutputBase** `IntegerOutputBase = number`

Integer output base.

**IsPrimeMillerRabinReps** `IsPrimeMillerRabinReps = number`

Number of extra Miller-Rabin tests to run on a number before declaring it a prime in IsPrime.

**LinePlotDrawLegends** `LinePlotDrawLegends = true`

Tells genius to draw the legends for line plotting functions such as LinePlot.

**LinePlotDrawAxisLabels** `LinePlotDrawAxisLabels = true`

Tells genius to draw the axis labels for line plotting functions such as LinePlot.

Version 1.0.16 onwards.

**LinePlotVariableNames** `LinePlotVariableNames = ["x","y","z","t"]`

Tells genius which variable names are used as default names for line plotting functions such as LinePlot and friends.

Version 1.0.10 onwards.

**LinePlotWindow** `LinePlotWindow = [x1,x2,y1,y2]`

Sets the limits for line plotting functions such as LinePlot.

**MaxDigits** `MaxDigits = number`

Maximum digits to display.

**MaxErrors** `MaxErrors = number`

Maximum errors to display.

**MixedFractions** `MixedFractions = boolean`

If true, mixed fractions are printed.

**NumericalIntegralFunction** `NumericalIntegralFunction = function`

The function used for numerical integration in NumericalIntegral.

**NumericalIntegralSteps** `NumericalIntegralSteps = number`

Steps to perform in NumericalIntegral.

**OutputChopExponent** `OutputChopExponent = number`

When another number in the object being printed (a matrix or a value) is greater than $10^{-\text{OutputChopWhenExponent}}$, and the number being printed is less than $10^{-\text{OutputChopExponent}}$, then display `0.0` instead of the number.

Output is never chopped if `OutputChopExponent` is zero. It must be a non-negative integer.

If you want output always chopped according to `OutputChopExponent`, then set `OutputChopWhenExponent`, to something greater than or equal to `OutputChopExponent`.

**OutputChopWhenExponent** `OutputChopWhenExponent = number`

When to chop output. See OutputChopExponent.

**OutputStyle** `OutputStyle = string`

Output style, this can be `normal`, `latex`, `mathml` or `troff`.

This affects mostly how matrices and fractions are printed out and is useful for pasting into documents. For example you can set this to the latex by:

```
OutputStyle = "latex"
```

**ResultsAsFloats** `ResultsAsFloats = boolean`

Convert all results to floats before printing.

**ScientificNotation** `ScientificNotation = boolean`

Use scientific notation.

**SlopefieldTicks** `SlopefieldTicks = [vertical,horizontal]`

Sets the number of vertical and horizontal ticks in a slopefield plot. (See SlopefieldPlot).

Version 1.0.10 onwards.

**SumProductNumberOfTries** `SumProductNumberOfTries = number`

How many iterations to try for InfiniteSum and InfiniteProduct.

**SumProductSFS** `SumProductSFS = number`

How many successive steps to be within tolerance for InfiniteSum and InfiniteProduct.

**SumProductTolerance** `SumProductTolerance = number`

Tolerance for InfiniteSum and InfiniteProduct.

**SurfacePlotDrawLegends** `SurfacePlotDrawLegends = true`

Tells genius to draw the legends for surface plotting functions such as SurfacePlot.

Version 1.0.16 onwards.

**SurfacePlotVariableNames** `SurfacePlotVariableNames = ["x","y","z"]`

Tells genius which variable names are used as default names for surface plotting functions using SurfacePlot. Note that the `z` does not refer to the dependent (vertical) axis, but to the independent complex variable **`z=x+iy`**.

Version 1.0.10 onwards.

**SurfacePlotWindow** `SurfacePlotWindow = [x1,x2,y1,y2,z1,z2]`

Sets the limits for surface plotting (See SurfacePlot).

**VectorfieldNormalized** `VectorfieldNormalized = true`

Should the vectorfield plotting have normalized arrow length. If true, vector fields will only show direction and not magnitude. (See VectorfieldPlot).

**VectorfieldTicks** `VectorfieldTicks = [vertical,horizontal]`

Sets the number of vertical and horizontal ticks in a vectorfield plot. (See VectorfieldPlot).

Version 1.0.10 onwards.

## 11.4 Constants

**CatalanConstant** `CatalanConstant`

Catalan's Constant, approximately 0.915... It is defined to be the series where terms are `(-1^k)/((2*k+1)^2)`, where `k` ranges from 0 to infinity.

See Wikipedia, or Mathworld for more information.

**EulerConstant** `EulerConstant`

Aliases: `gamma`

Euler's constant gamma. Sometimes called the Euler-Mascheroni constant.

See Wikipedia or Planetmath or Mathworld for more information.

**GoldenRatio** `GoldenRatio`

The Golden Ratio.

See Wikipedia or Planetmath or Mathworld for more information.

**Gravity** `Gravity`

Free fall acceleration at sea level in meters per second squared. This is the standard gravity constant 9.80665. The gravity in your particular neck of the woods might be different due to different altitude and the fact that the earth is not perfectly round and uniform.

See Wikipedia for more information.

**e** `e`

The base of the natural logarithm. `e^x` is the exponential function exp. It is approximately 2.71828182846... This number is sometimes called Euler's number, although there are several numbers that are also called Euler's. An example is the gamma constant: EulerConstant.

See Wikipedia or Planetmath or Mathworld for more information.

**pi** `pi`

The number pi, that is the ratio of a circle's circumference to its diameter. This is approximately 3.14159265359...

See Wikipedia or Planetmath or Mathworld for more information.

## 11.5 Numeric

**AbsoluteValue** `AbsoluteValue (x)`

Aliases: `abs`

Absolute value of a number and if `x` is a complex value the modulus of `x`. I.e. this the distance of `x` to the origin. This is equivalent to `|x|`.

See Wikipedia, Planetmath (absolute value), Planetmath (modulus), Mathworld (absolute value) or Mathworld (complex modulus) for more information.

**Chop** `Chop (x)`

Replace very small number with zero.

**ComplexConjugate** `ComplexConjugate (z)`

Aliases: `conj Conj`

Calculates the complex conjugate of the complex number `z`. If `z` is a vector or matrix, all its elements are conjugated.

See Wikipedia for more information.

**Denominator** `Denominator (x)`

Get the denominator of a rational number.

See Wikipedia for more information.

**FractionalPart** `FractionalPart (x)`

Return the fractional part of a number.

See Wikipedia for more information.

**Im** `Im (z)`

Aliases: `ImaginaryPart`

Get the imaginary part of a complex number. For example **`Re(3+4i)`** yields 4.

See Wikipedia for more information.

**IntegerQuotient** `IntegerQuotient (m,n)`

Division without remainder.

**IsComplex** `IsComplex (num)`

Check if argument is a complex (non-real) number. Do note that we really mean nonreal number. That is, **`IsComplex(3)`** yields false, while **`IsComplex(3-1i)`** yields true.

**IsComplexRational** `IsComplexRational (num)`

Check if argument is a possibly complex rational number. That is, if both real and imaginary parts are given as rational numbers. Of course rational simply means "not stored as a floating point number."

**IsFloat** `IsFloat (num)`

Check if argument is a real floating point number (non-complex).

**IsGaussInteger** `IsGaussInteger (num)`

Aliases: `IsComplexInteger`

Check if argument is a possibly complex integer. That is a complex integer is a number of the form **`n+1i*m`** where `n` and `m` are integers.

**IsInteger** `IsInteger (num)`

Check if argument is an integer (non-complex).

**IsNonNegativeInteger** `IsNonNegativeInteger (num)`

Check if argument is a non-negative real integer. That is, either a positive integer or zero.

**IsPositiveInteger** `IsPositiveInteger (num)`

Aliases: `IsNaturalNumber`

Check if argument is a positive real integer. Note that we accept the convention that 0 is not a natural number.

**IsRational** `IsRational (num)`

Check if argument is a rational number (non-complex). Of course rational simply means "not stored as a floating point number."

**IsReal** `IsReal (num)`

Check if argument is a real number.

**Numerator** `Numerator (x)`

Get the numerator of a rational number.

See Wikipedia for more information.

**Re** `Re (z)`

Aliases: `RealPart`

Get the real part of a complex number. For example **`Re(3+4i)`** yields 3.

See Wikipedia for more information.

**Sign** `Sign (x)`

Aliases: `sign`

Return the sign of a number. That is returns `-1` if value is negative, `0` if value is zero and `1` if value is positive. If `x` is a complex value then `Sign` returns the direction or 0.

**ceil** `ceil (x)`

Aliases: `Ceiling`

Get the lowest integer more than or equal to `n`. Examples:

```
genius> ceil(1.1)
= 2
genius> ceil(-1.1)
= -1
```

Note that you should be careful and notice that floating point numbers are stored an binary and so may not be what you expect. For example **`ceil(420/4.2)`** returns 101 instead of the expected 100. This is because 4.2 is actually very slightly less than 4.2. Use rational representation **`42/10`** if you want exact arithmetic.

**exp** `exp (x)`

The exponential function. This is the function **`e^x`** where `e` is the base of the natural logarithm.

See Wikipedia or Planetmath or Mathworld for more information.

**float** `float (x)`

Make number a floating point value. That is returns the floating point representation of the number `x`.

**floor** `floor (x)`

Aliases: `Floor`

Get the highest integer less than or equal to `n`.

**ln** `ln (x)`

The natural logarithm, the logarithm to base `e`.

See Wikipedia or Planetmath or Mathworld for more information.

**log** `log (x)`

`log (x,b)`

Logarithm of `x` base `b` (calls DiscreteLog if in modulo mode), if base is not given, e is used.

**log10** `log10 (x)`

Logarithm of `x` base 10.

**log2** `log2 (x)`

Aliases: `lg`

Logarithm of `x` base 2.

**max** `max (a,args...)`

Aliases: `Max Maximum`

Returns the maximum of arguments or matrix.

**min** `min (a,args...)`

Aliases: `Min Minimum`

Returns the minimum of arguments or matrix.

**rand** `rand (size...)`

Generate random float in the range `[0,1)`. If size is given then a matrix (if two numbers are specified) or vector (if one number is specified) of the given size returned.

**randint** `randint (max,size...)`

Generate random integer in the range `[0,max)`. If size is given then a matrix (if two numbers are specified) or vector (if one number is specified) of the given size returned. For example,

```
genius> randint(4)
= 3
genius> randint(4,2)
=
[0      1]
genius> randint(4,2,3)
=
[2      2       1
 0      0       3]
```

**round** `round (x)`

Aliases: `Round`

Round a number.

**sqrt** `sqrt (x)`

Aliases: `SquareRoot`

The square root. When operating modulo some integer will return either a `null` or a vector of the square roots. Examples:

```
genius> sqrt(2)
= 1.41421356237
genius> sqrt(-1)
= 1i
genius> sqrt(4) mod 7
=
[2      5]
genius> 2*2 mod 7
= 4
```

See Planetmath for more information.

**trunc** `trunc (x)`

Aliases: `Truncate IntegerPart`

Truncate number to an integer (return the integer part).

## 11.6 Trigonometry

**acos** `acos (x)`

Aliases: `arccos`

The arccos (inverse cos) function.

**acosh** `acosh (x)`

Aliases: `arccosh`

The arccosh (inverse cosh) function.

**acot** `acot (x)`

Aliases: `arccot`

The arccot (inverse cot) function.

**acoth** `acoth (x)`

Aliases: `arccoth`

The arccoth (inverse coth) function.

**acsc** `acsc (x)`

Aliases: `arccsc`

The inverse cosecant function.

**acsch** `acsch (x)`

Aliases: `arccsch`

The inverse hyperbolic cosecant function.

**asec** `asec (x)`

Aliases: `arcsec`

The inverse secant function.

**asech** `asech (x)`

Aliases: `arcsech`

The inverse hyperbolic secant function.

**asin** `asin (x)`

Aliases: `arcsin`

The arcsin (inverse sin) function.

**asinh** `asinh (x)`

Aliases: `arcsinh`

The arcsinh (inverse sinh) function.

**atan** `atan (x)`

Aliases: `arctan`

Calculates the arctan (inverse tan) function.

See Wikipedia or Mathworld for more information.

**atanh** `atanh (x)`

Aliases: `arctanh`

The arctanh (inverse tanh) function.

**atan2** `atan2 (y, x)`

Aliases: `arctan2`

Calculates the arctan2 function. If **`x>0`** then it returns **`atan(y/x)`**. If **`x<0`** then it returns **`sign(y) * (pi - atan(|y/x|)`**. When **`x=0`** it returns **`sign(y) * pi/2`**. **`atan2(0,0)`** returns 0 rather then failing.

See Wikipedia or Mathworld for more information.

**cos** `cos (x)`

Calculates the cosine function.

See Planetmath for more information.

**cosh** `cosh (x)`

Calculates the hyperbolic cosine function.

See Planetmath for more information.

**cot** `cot (x)`

The cotangent function.

See Planetmath for more information.

**coth** `coth (x)`

The hyperbolic cotangent function.

See Planetmath for more information.

**csc** `csc (x)`

The cosecant function.

See Planetmath for more information.

**csch** `csch (x)`

The hyperbolic cosecant function.

See Planetmath for more information.

**sec** `sec (x)`

The secant function.

See Planetmath for more information.

**sech** `sech (x)`

The hyperbolic secant function.

See Planetmath for more information.

**sin** `sin (x)`

Calculates the sine function.

See Planetmath for more information.

**sinh** `sinh (x)`

Calculates the hyperbolic sine function.

See Planetmath for more information.

**tan** `tan (x)`

Calculates the tan function.

See Planetmath for more information.

**tanh** `tanh (x)`

The hyperbolic tangent function.

See Planetmath for more information.

## 11.7 Number Theory

**AreRelativelyPrime** `AreRelativelyPrime (a,b)`

Are the real integers `a` and `b` relatively prime? Returns `true` or `false`.

See Planetmath or Mathworld for more information.

**BernoulliNumber** `BernoulliNumber (n)`

Return the `n`th Bernoulli number.

See Wikipedia or Mathworld for more information.

**ChineseRemainder** `ChineseRemainder (a,m)`

Aliases: `CRT`

Find the `x` that solves the system given by the vector `a` and modulo the elements of `m`, using the Chinese Remainder Theorem.

See Wikipedia or Planetmath or Mathworld for more information.

**CombineFactorizations** `CombineFactorizations (a,b)`

Given two factorizations, give the factorization of the product.

See Factorize.

**ConvertFromBase** `ConvertFromBase (v,b)`

Convert a vector of values indicating powers of b to a number.

**ConvertToBase** `ConvertToBase (n,b)`

Convert a number to a vector of powers for elements in base `b`.

**DiscreteLog** `DiscreteLog (n,b,q)`

Find discrete log of `n` base `b` in $F_q$, the finite field of order `q`, where `q` is a prime, using the Silver-Pohlig-Hellman algorithm.

See Wikipedia or Planetmath or Mathworld for more information.

**Divides** `Divides (m,n)`

Checks divisibility (if `m` divides `n`).

**EulerPhi** `EulerPhi (n)`

Compute the Euler phi function for `n`, that is the number of integers between 1 and `n` relatively prime to `n`.

See Wikipedia or Planetmath or Mathworld for more information.

**ExactDivision** `ExactDivision (n,d)`

Return **`n/d`** but only if `d` divides `n`. If `d` does not divide `n` then this function returns garbage. This is a lot faster for very large numbers than the operation **`n/d`**, but of course only useful if you know that the division is exact.

**Factorize** `Factorize (n)`

Return factorization of a number as a matrix. The first row is the primes in the factorization (including 1) and the second row are the powers. So for example:

```
genius> Factorize(11*11*13)
=
[1       11       13
 1       2        1]
```

See Wikipedia for more information.

**Factors** `Factors (n)`

Return all factors of `n` in a vector. This includes all the non-prime factors as well. It includes 1 and the number itself. So for example to print all the perfect numbers (those that are sums of their factors) up to the number 1000 you could do (this is of course very inefficient)

```
for n=1 to 1000 do (
    if MatrixSum (Factors(n)) == 2*n then
        print(n)
)
```

**FermatFactorization** `FermatFactorization (n,tries)`

Attempt Fermat factorization of `n` into **`(t-s)*(t+s)`**, returns `t` and `s` as a vector if possible, `null` otherwise. `tries` specifies the number of tries before giving up.

This is a fairly good factorization if your number is the product of two factors that are very close to each other.

See Wikipedia for more information.

**FindPrimitiveElementMod** `FindPrimitiveElementMod (q)`

Find the first primitive element in $F_q$, the finite group of order `q`. Of course `q` must be a prime.

**FindRandomPrimitiveElementMod** `FindRandomPrimitiveElementMod (q)`

Find a random primitive element in $F_q$, the finite group of order `q` (q must be a prime).

**IndexCalculus** `IndexCalculus (n,b,q,S)`

Compute discrete log base `b` of n in $F_q$, the finite group of order `q` (`q` a prime), using the factor base `S`. `S` should be a column of primes possibly with second column precalculated by IndexCalculusPrecalculation.

**IndexCalculusPrecalculation** `IndexCalculusPrecalculation (b,q,S)`

Run the precalculation step of IndexCalculus for logarithms base `b` in $F_q$, the finite group of order `q` (`q` a prime), for the factor base `S` (where `S` is a column vector of primes). The logs will be precalculated and returned in the second column.

**IsEven** `IsEven (n)`

Tests if an integer is even.

**IsMersennePrimeExponent** `IsMersennePrimeExponent (p)`

Tests if a positive integer `p` is a Mersenne prime exponent. That is if $2^p$-1 is a prime. It does this by looking it up in a table of known values, which is relatively short. See also MersennePrimeExponents and LucasLehmer.

See Wikipedia, Planetmath, Mathworld or GIMPS for more information.

**IsNthPower** `IsNthPower (m,n)`

Tests if a rational number `m` is a perfect `n`th power. See also IsPerfectPower and IsPerfectSquare.

**IsOdd** `IsOdd (n)`

Tests if an integer is odd.

**IsPerfectPower** `IsPerfectPower (n)`

Check an integer is any perfect power, $a^b$.

**IsPerfectSquare** `IsPerfectSquare (n)`

Check an integer for being a perfect square of an integer. The number must be a real integer. Negative integers are of course never perfect squares of real integers.

**IsPrime** `IsPrime (n)`

Tests primality of integers, for numbers less than 2.5e10 the answer is deterministic (if Riemann hypothesis is true). For numbers larger, the probability of a false positive depends on IsPrimeMillerRabinReps. That is the probability of false positive is 1/4 to the power `IsPrimeMillerRabinReps`. The default value of 22 yields a probability of about 5.7e-14.

If `false` is returned, you can be sure that the number is a composite. If you want to be absolutely sure that you have a prime you can use MillerRabinTestSure but it may take a lot longer.

See Planetmath or Mathworld for more information.

**IsPrimitiveMod** `IsPrimitiveMod (g,q)`

Check if `g` is primitive in $F_q$, the finite group of order `q`, where `q` is a prime. If `q` is not prime results are bogus.

**IsPrimitiveModWithPrimeFactors** `IsPrimitiveModWithPrimeFactors (g,q,f)`

Check if `g` is primitive in $F_q$, the finite group of order `q`, where `q` is a prime and `f` is a vector of prime factors of `q`-1. If `q` is not prime results are bogus.

**IsPseudoprime** `IsPseudoprime (n,b)`

If `n` is a pseudoprime base `b` but not a prime, that is if **`b^(n-1) == 1 mod n`**. This calls the PseudoprimeTest

**IsStrongPseudoprime** `IsStrongPseudoprime (n,b)`

Test if `n` is a strong pseudoprime to base `b` but not a prime.

**Jacobi** `Jacobi (a,b)`

Aliases: `JacobiSymbol`

Calculate the Jacobi symbol (a/b) (b should be odd).

**JacobiKronecker** `JacobiKronecker (a,b)`

Aliases: `JacobiKroneckerSymbol`

Calculate the Jacobi symbol (a/b) with the Kronecker extension (a/2)=(2/a) when a odd, or (a/2)=0 when a even.

**LeastAbsoluteResidue** `LeastAbsoluteResidue (a,n)`

Return the residue of `a` mod `n` with the least absolute value (in the interval -n/2 to n/2).

**Legendre** `Legendre (a,p)`

Aliases: `LegendreSymbol`

Calculate the Legendre symbol (a/p).

See Planetmath or Mathworld for more information.

**LucasLehmer** `LucasLehmer (p)`

Test if $2^p$-1 is a Mersenne prime using the Lucas-Lehmer test. See also MersennePrimeExponents and IsMersennePrimeExponent.

See Wikipedia or Planetmath or Mathworld for more information.

**LucasNumber** `LucasNumber (n)`

Returns the `n`th Lucas number.

See Wikipedia or Planetmath or Mathworld for more information.

**MaximalPrimePowerFactors** `MaximalPrimePowerFactors (n)`

Return all maximal prime power factors of a number.

**MersennePrimeExponents** `MersennePrimeExponents`

A vector of known Mersenne prime exponents, that is a list of positive integers `p` such that $2^p$-1 is a prime. See also IsMersennePrimeExponent and LucasLehmer.

See Wikipedia, Planetmath, Mathworld or GIMPS for more information.

**MillerRabinTest** `MillerRabinTest (n,reps)`

Use the Miller-Rabin primality test on `n`, `reps` number of times. The probability of false positive is **`(1/4)^reps`**. It is probably usually better to use IsPrime since that is faster and better on smaller integers.

See Wikipedia or Planetmath or Mathworld for more information.

**MillerRabinTestSure** `MillerRabinTestSure (n)`

Use the Miller-Rabin primality test on `n` with enough bases that assuming the Generalized Riemann Hypothesis the result is deterministic.

See Wikipedia or Planetmath or Mathworld for more information.

**ModInvert** `ModInvert (n,m)`

Returns inverse of n mod m.

See Mathworld for more information.

**MoebiusMu** `MoebiusMu (n)`

Return the Moebius mu function evaluated in `n`. That is, it returns 0 if `n` is not a product of distinct primes and **`(-1)^k`** if it is a product of `k` distinct primes.

See Planetmath or Mathworld for more information.

**NextPrime** `NextPrime (n)`

Returns the least prime greater than `n`. Negatives of primes are considered prime and so to get the previous prime you can use **`-NextPrime(-n)`**.

This function uses the GMPs `mpz_nextprime`, which in turn uses the probabilistic Miller-Rabin test (See also MillerRabinTest). The probability of false positive is not tunable, but is low enough for all practical purposes.

See Planetmath or Mathworld for more information.

**PadicValuation** `PadicValuation (n,p)`

Returns the p-adic valuation (number of trailing zeros in base `p`).

See Planetmath for more information.

**PowerMod** `PowerMod (a,b,m)`

Compute **`a^b mod m`**. The `b`'s power of `a` modulo `m`. It is not necessary to use this function as it is automatically used in modulo mode. Hence **`a^b mod m`** is just as fast.

**Prime** `Prime (n)`

Aliases: `prime`

Return the `n`th prime (up to a limit).

See Planetmath or Mathworld for more information.

**PrimeFactors** `PrimeFactors (n)`

Return all prime factors of a number as a vector.

See Mathworld for more information.

**PseudoprimeTest** `PseudoprimeTest (n,b)`

Pseudoprime test, returns `true` if and only if **`b^(n-1) == 1 mod n`**

See Planetmath or Mathworld for more information.

**RemoveFactor** `RemoveFactor (n,m)`

Removes all instances of the factor `m` from the number `n`. That is divides by the largest power of `m`, that divides `n`.

See Planetmath or Mathworld for more information.

**SilverPohligHellmanWithFactorization** `SilverPohligHellmanWithFactorization (n,b,q,f)`

Find discrete log of `n` base `b` in $F_q$, the finite group of order `q`, where `q` is a prime using the Silver-Pohlig-Hellman algorithm, given `f` being the factorization of `q`-1.

**SqrtModPrime** `SqrtModPrime (n,p)`

Find square root of `n` modulo `p` (where `p` is a prime). Null is returned if not a quadratic residue.

See Planetmath or Mathworld for more information.

**StrongPseudoprimeTest** `StrongPseudoprimeTest (n,b)`

Run the strong pseudoprime test base `b` on `n`.

See Planetmath or Mathworld for more information.

**gcd** `gcd (a,args...)`

Aliases: `GCD`

Greatest common divisor of integers. You can enter as many integers in the argument list, or you can give a vector or a matrix of integers. If you give more than one matrix of the same size then GCD is done element by element.

See Planetmath or Mathworld for more information.

**lcm** `lcm (a,args...)`

Aliases: `LCM`

Least common multiplier of integers. You can enter as many integers in the argument list, or you can give a vector or a matrix of integers. If you give more than one matrix of the same size then LCM is done element by element.

See Planetmath or Mathworld for more information.

## 11.8 Matrix Manipulation

**ApplyOverMatrix** `ApplyOverMatrix (a,func)`

Apply a function over all entries of a matrix and return a matrix of the results.

**ApplyOverMatrix2** `ApplyOverMatrix2 (a,b,func)`

Apply a function over all entries of 2 matrices (or 1 value and 1 matrix) and return a matrix of the results.

**ColumnsOf** `ColumnsOf (M)`

Gets the columns of a matrix as a horizontal vector.

**ComplementSubmatrix** `ComplementSubmatrix (m,r,c)`

Remove column(s) and row(s) from a matrix.

**CompoundMatrix** `CompoundMatrix (k,A)`

Calculate the kth compound matrix of A.

**CountZeroColumns** `CountZeroColumns (M)`

Count the number of zero columns in a matrix. For example once your column reduce a matrix you can use this to find the nullity. See cref and Nullity.

**DeleteColumn** `DeleteColumn (M,col)`

Delete a column of a matrix.

**DeleteRow** `DeleteRow (M,row)`

Delete a row of a matrix.

**DiagonalOf** `DiagonalOf (M)`

Gets the diagonal entries of a matrix as a column vector.

See Wikipedia for more information.

**DotProduct** `DotProduct (u,v)`

Get the dot product of two vectors. The vectors must be of the same size. No conjugates are taken so this is a bilinear form even if working over the complex numbers.

See Planetmath for more information.

**ExpandMatrix** `ExpandMatrix (M)`

Expands a matrix just like we do on unquoted matrix input. That is we expand any internal matrices as blocks. This is a way to construct matrices out of smaller ones and this is normally done automatically on input unless the matrix is quoted.

**HermitianProduct** `HermitianProduct (u,v)`

Aliases: `InnerProduct`

Get the Hermitian product of two vectors. The vectors must be of the same size. This is a sesquilinear form using the identity matrix.

See Mathworld for more information.

**I** `I (n)`

Aliases: `eye`

Return an identity matrix of a given size, that is `n` by `n`. If `n` is zero, returns `null`.

See Planetmath for more information.

**IndexComplement** `IndexComplement (vec,msize)`

Return the index complement of a vector of indexes. Everything is one based. For example for vector **`[2,3]`** and size **`5`**, we return **`[1,4,5]`**. If `msize` is 0, we always return `null`.

**IsDiagonal** `IsDiagonal (M)`

Is a matrix diagonal.

See Wikipedia or Planetmath for more information.

**IsIdentity** `IsIdentity (x)`

Check if a matrix is the identity matrix. Automatically returns `false` if the matrix is not square. Also works on numbers, in which case it is equivalent to **`x==1`**. When `x` is `null` (we could think of that as a 0 by 0 matrix), no error is generated and `false` is returned.

**IsLowerTriangular** `IsLowerTriangular (M)`

Is a matrix lower triangular. That is, are all the entries above the diagonal zero.

**IsMatrixInteger** `IsMatrixInteger (M)`

Check if a matrix is a matrix of an integers (non-complex).

**IsMatrixNonnegative** `IsMatrixNonnegative (M)`

Check if a matrix is non-negative, that is if each element is non-negative. Do not confuse positive matrices with positive semi-definite matrices.

See Wikipedia for more information.

**IsMatrixPositive** `IsMatrixPositive (M)`

Check if a matrix is positive, that is if each element is positive (and hence real). In particular, no element is 0. Do not confuse positive matrices with positive definite matrices.

See Wikipedia for more information.

**IsMatrixRational** `IsMatrixRational (M)`

Check if a matrix is a matrix of rational (non-complex) numbers.

**IsMatrixReal** `IsMatrixReal (M)`

Check if a matrix is a matrix of real (non-complex) numbers.

**IsMatrixSquare** `IsMatrixSquare (M)`

Check if a matrix is square, that is its width is equal to its height.

**IsUpperTriangular** `IsUpperTriangular (M)`

Is a matrix upper triangular? That is, a matrix is upper triangular if all the entries below the diagonal are zero.

**IsValueOnly** `IsValueOnly (M)`

Check if a matrix is a matrix of numbers only. Many internal functions make this check. Values can be any number including complex numbers.

**IsVector** `IsVector (v)`

Is argument a horizontal or a vertical vector. Genius does not distinguish between a matrix and a vector and a vector is just a 1 by `n` or `n` by 1 matrix.

**IsZero** `IsZero (x)`

Check if a matrix is composed of all zeros. Also works on numbers, in which case it is equivalent to **`x==0`**. When `x` is `null` (we could think of that as a 0 by 0 matrix), no error is generated and `true` is returned as the condition is vacuous.

**LowerTriangular** `LowerTriangular (M)`

Returns a copy of the matrix `M` with all the entries above the diagonal set to zero.

**MakeDiagonal** `MakeDiagonal (v,arg...)`

Aliases: `diag`

Make diagonal matrix from a vector. Alternatively you can pass in the values to put on the diagonal as arguments. So **`MakeDiagonal([1,2,3])`** is the same as **`MakeDiagonal(1,2,3)`**.

See Wikipedia or Planetmath for more information.

**MakeVector** `MakeVector (A)`

Make column vector out of matrix by putting columns above each other. Returns `null` when given `null`.

**MatrixProduct** `MatrixProduct (A)`

Calculate the product of all elements in a matrix or vector. That is we multiply all the elements and return a number that is the product of all the elements.

**MatrixSum** `MatrixSum (A)`

Calculate the sum of all elements in a matrix or vector. That is we add all the elements and return a number that is the sum of all the elements.

**MatrixSumSquares** `MatrixSumSquares (A)`

Calculate the sum of squares of all elements in a matrix or vector.

**NonzeroColumns** `NonzeroColumns (M)`

Returns a row vector of the indices of nonzero columns in the matrix `M`.

Version 1.0.18 onwards.

**NonzeroElements** `NonzeroElements (v)`

Returns a row vector of the indices of nonzero elements in the vector `v`.

Version 1.0.18 onwards.

**OuterProduct** `OuterProduct (u,v)`

Get the outer product of two vectors. That is, suppose that `u` and `v` are vertical vectors, then the outer product is **`v * u.'`**.

**ReverseVector** `ReverseVector (v)`

Reverse elements in a vector. Return `null` if given `null`

**RowSum** `RowSum (m)`

Calculate sum of each row in a matrix and return a vertical vector with the result.

**RowSumSquares** `RowSumSquares (m)`

Calculate sum of squares of each row in a matrix and return a vertical vector with the results.

**RowsOf** `RowsOf (M)`

Gets the rows of a matrix as a vertical vector. Each element of the vector is a horizontal vector that is the corresponding row of `M`. This function is useful if you wish to loop over the rows of a matrix. For example, as **`for r in RowsOf(M) do something(r)`**.

**SetMatrixSize** `SetMatrixSize (M,rows,columns)`

Make new matrix of given size from old one. That is, a new matrix will be returned to which the old one is copied. Entries that don't fit are clipped and extra space is filled with zeros. If `rows` or `columns` are zero then `null` is returned.

**ShuffleVector** `ShuffleVector (v)`

Shuffle elements in a vector. Return `null` if given `null`.

Version 1.0.13 onwards.

**SortVector** `SortVector (v)`

Sort vector elements in an increasing order.

**StripZeroColumns** `StripZeroColumns (M)`

Removes any all-zero columns of `M`.

**StripZeroRows** `StripZeroRows (M)`

Removes any all-zero rows of `M`.

**Submatrix** `Submatrix (m,r,c)`

Return column(s) and row(s) from a matrix. This is just equivalent to **`m@(r,c)`**. `r` and `c` should be vectors of rows and columns (or single numbers if only one row or column is needed).

**SwapRows** `SwapRows (m,row1,row2)`

Swap two rows in a matrix.

**UpperTriangular** `UpperTriangular (M)`

Returns a copy of the matrix `M` with all the entries below the diagonal set to zero.

**columns** `columns (M)`

Get the number of columns of a matrix.

**elements** `elements (M)`

Get the total number of elements of a matrix. This is the number of columns times the number of rows.

**ones** `ones (rows,columns...)`

Make an matrix of all ones (or a row vector if only one argument is given). Returns `null` if either rows or columns are zero.

**rows** `rows (M)`

Get the number of rows of a matrix.

**zeros** `zeros (rows,columns...)`

Make a matrix of all zeros (or a row vector if only one argument is given). Returns `null` if either rows or columns are zero.

## 11.9 Linear Algebra

**AuxiliaryUnitMatrix** `AuxiliaryUnitMatrix (n)`

Get the auxiliary unit matrix of size `n`. This is a square matrix with that is all zero except the superdiagonal being all ones. It is the Jordan block matrix of one zero eigenvalue.

See Planetmath or Mathworld for more information on Jordan Canonical Form.

**BilinearForm** `BilinearForm (v,A,w)`

Evaluate (v,w) with respect to the bilinear form given by the matrix A.

**BilinearFormFunction** `BilinearFormFunction (A)`

Return a function that evaluates two vectors with respect to the bilinear form given by A.

**CharacteristicPolynomial** `CharacteristicPolynomial (M)`

Aliases: `CharPoly`

Get the characteristic polynomial as a vector. That is, return the coefficients of the polynomial starting with the constant term. This is the polynomial defined by **`det(M-xI)`**. The roots of this polynomial are the eigenvalues of `M`. See also CharacteristicPolynomialFunction.

See Planetmath for more information.

**CharacteristicPolynomialFunction** `CharacteristicPolynomialFunction (M)`

Get the characteristic polynomial as a function. This is the polynomial defined by **`det(M-xI)`**. The roots of this polynomial are the eigenvalues of `M`. See also CharacteristicPolynomial.

See Planetmath for more information.

**ColumnSpace** `ColumnSpace (M)`

Get a basis matrix for the columnspace of a matrix. That is, return a matrix whose columns are the basis for the column space of `M`. That is the space spanned by the columns of `M`.

**CommutationMatrix** `CommutationMatrix (m, n)`

Return the commutation matrix **`K(m,n)`**, which is the unique **`m*n`** by **`m*n`** matrix such that **`K(m,n) * MakeVector(A) = MakeVector(A.')`** for all `m` by `n` matrices `A`.

**CompanionMatrix** `CompanionMatrix (p)`

Companion matrix of a polynomial (as vector).

**ConjugateTranspose** `ConjugateTranspose (M)`

Conjugate transpose of a matrix (adjoint). This is the same as the `'` operator.

See Planetmath for more information.

**Convolution** `Convolution (a,b)`

Aliases: `convol`

Calculate convolution of two horizontal vectors.

**ConvolutionVector** `ConvolutionVector (a,b)`

Calculate convolution of two horizontal vectors. Return result as a vector and not added together.

**CrossProduct** `CrossProduct (v,w)`

CrossProduct of two vectors in $R^3$ as a column vector.

**DeterminantalDivisorsInteger** `DeterminantalDivisorsInteger (M)`

Get the determinantal divisors of an integer matrix (not its characteristic).

**DirectSum** `DirectSum (M,N...)`

Direct sum of matrices.

**DirectSumMatrixVector** `DirectSumMatrixVector (v)`

Direct sum of a vector of matrices.

**Eigenvalues** `Eigenvalues (M)`

Aliases: `eig`

Get the eigenvalues of a square matrix. Currently only works for matrices of size up to 4 by 4, or for triangular matrices (for which the eigenvalues are on the diagonal).

See Wikipedia or Planetmath or Mathworld for more information.

**Eigenvectors** `Eigenvectors (M)`

```
Eigenvectors (M, &eigenvalues)
```

```
Eigenvectors (M, &eigenvalues, &multiplicities)
```

Get the eigenvectors of a square matrix. Optionally get also the eigenvalues and their algebraic multiplicities. Currently only works for matrices of size up to 2 by 2.

See Wikipedia or Planetmath or Mathworld for more information.

**GramSchmidt** `GramSchmidt (v,B...)`

Apply the Gram-Schmidt process (to the columns) with respect to inner product given by `B`. If `B` is not given then the standard Hermitian product is used. `B` can either be a sesquilinear function of two arguments or it can be a matrix giving a sesquilinear form. The vectors will be made orthonormal with respect to `B`.

See Planetmath for more information.

**HankelMatrix** `HankelMatrix (c,r)`

Hankel matrix.

**HilbertMatrix** `HilbertMatrix (n)`

Hilbert matrix of order `n`.

See Planetmath for more information.

**Image** `Image (T)`

Get the image (columnspace) of a linear transform.

**InfNorm** `InfNorm (v)`

Get the Inf Norm of a vector, sometimes called the sup norm or the max norm.

**InvariantFactorsInteger** `InvariantFactorsInteger (M)`

Get the invariant factors of a square integer matrix (not its characteristic).

**InverseHilbertMatrix** `InverseHilbertMatrix (n)`

Inverse Hilbert matrix of order `n`.

See Planetmath for more information.

**IsHermitian** `IsHermitian (M)`

Is a matrix Hermitian. That is, is it equal to its conjugate transpose.

See Planetmath for more information.

**IsInSubspace** `IsInSubspace (v,W)`

Test if a vector is in a subspace.

**IsInvertible** `IsInvertible (n)`

Is a matrix (or number) invertible (Integer matrix is invertible if and only if it is invertible over the integers).

**IsInvertibleField** `IsInvertibleField (n)`

Is a matrix (or number) invertible over a field.

**IsNormal** `IsNormal (M)`

Is `M` a normal matrix. That is, does **`M*M' == M'*M`**.

See Planetmath or Mathworld for more information.

**IsPositiveDefinite** `IsPositiveDefinite (M)`

Is `M` a Hermitian positive definite matrix. That is if **`HermitianProduct(M*v,v)`** is always strictly positive for any vector `v`. `M` must be square and Hermitian to be positive definite. The check that is performed is that every principal submatrix has a non-negative determinant. (See HermitianProduct)

Note that some authors (for example Mathworld) do not require that `M` be Hermitian, and then the condition is on the real part of the inner product, but we do not take this view. If you wish to perform this check, just check the Hermitian part of the matrix `M` as follows: **`IsPositiveDefinite(M+M')`**.

See Planetmath or Mathworld for more information.

**IsPositiveSemidefinite** `IsPositiveSemidefinite (M)`

Is `M` a Hermitian positive semidefinite matrix. That is if **`HermitianProduct(M*v,v)`** is always non-negative for any vector `v`. `M` must be square and Hermitian to be positive semidefinite. The check that is performed is that every principal submatrix has a non-negative determinant. (See HermitianProduct)

Note that some authors do not require that `M` be Hermitian, and then the condition is on the real part of the inner product, but we do not take this view. If you wish to perform this check, just check the Hermitian part of the matrix `M` as follows: **`IsPositiveSemidefinite(M+M')`**.

See Planetmath or Mathworld for more information.

**IsSkewHermitian** `IsSkewHermitian (M)`

Is a matrix skew-Hermitian. That is, is the conjugate transpose equal to negative of the matrix.

See Planetmath for more information.

**IsUnitary** `IsUnitary (M)`

Is a matrix unitary? That is, does **`M'*M`** and **`M*M'`** equal the identity.

See Planetmath or Mathworld for more information.

**JordanBlock** `JordanBlock (n,lambda)`

Aliases: `J`

Get the Jordan block corresponding to the eigenvalue `lambda` with multiplicity `n`.

See Planetmath or Mathworld for more information.

**Kernel** `Kernel (T)`

Get the kernel (nullspace) of a linear transform.

(See NullSpace)

**KroneckerProduct** `KroneckerProduct (M, N)`

Aliases: `TensorProduct`

Compute the Kronecker product (tensor product in standard basis) of two matrices.

See Wikipedia, Planetmath or Mathworld for more information.

Version 1.0.18 onwards.

**LUDecomposition** `LUDecomposition (A, L, U)`

Get the LU decomposition of `A`, that is find a lower triangular matrix and upper triangular matrix whose product is `A` Store the result in the `L` and `U`, which should be references. It returns `true` if successful. For example suppose that A is a square matrix, then after running:

```
genius> LUDecomposition(A,&L,&U)
```

You will have the lower matrix stored in a variable called `L` and the upper matrix in a variable called `U`.

This is the LU decomposition of a matrix aka Crout and/or Cholesky reduction. (ISBN 0-201-11577-8 pp.99-103) The upper triangular matrix features a diagonal of values 1 (one). This is not Doolittle's Method, which features the 1's diagonal on the lower matrix.

Not all matrices have LU decompositions, for example `[0,1;1,0]` does not and this function returns `false` in this case and sets `L` and `U` to `null`.

See Wikipedia, Planetmath or Mathworld for more information.

**Minor** `Minor (M,i,j)`

Get the `i-j` minor of a matrix.

See Planetmath for more information.

**NonPivotColumns** `NonPivotColumns (M)`

Return the columns that are not the pivot columns of a matrix.

**Norm** `Norm (v,p...)`

Aliases: `norm`

Get the p Norm (or 2 Norm if no p is supplied) of a vector.

**NullSpace** `NullSpace (T)`

Get the nullspace of a matrix. That is the kernel of the linear mapping that the matrix represents. This is returned as a matrix whose column space is the nullspace of `T`.

See Planetmath for more information.

**Nullity** `Nullity (M)`

Aliases: `nullity`

Get the nullity of a matrix. That is, return the dimension of the nullspace; the dimension of the kernel of `M`.

See Planetmath for more information.

**OrthogonalComplement** `OrthogonalComplement (M)`

Get the orthogonal complement of the columnspace.

**PivotColumns** `PivotColumns (M)`

Return pivot columns of a matrix, that is columns that have a leading 1 in row reduced form. Also returns the row where they occur.

**Projection** `Projection (v,W,B...)`

Projection of vector `v` onto subspace `W` with respect to inner product given by `B`. If `B` is not given then the standard Hermitian product is used. `B` can either be a sesquilinear function of two arguments or it can be a matrix giving a sesquilinear form.

**QRDecomposition** `QRDecomposition (A, Q)`

Get the QR decomposition of a square matrix `A`, returns the upper triangular matrix `R` and sets `Q` to the orthogonal (unitary) matrix. `Q` should be a reference or `null` if you don't want any return. For example:

```
genius> R = QRDecomposition(A,&Q)
```

You will have the upper triangular matrix stored in a variable called `R` and the orthogonal (unitary) matrix stored in `Q`.

See Wikipedia or Planetmath or Mathworld for more information.

**RayleighQuotient** `RayleighQuotient (A,x)`

Return the Rayleigh quotient (also called the Rayleigh-Ritz quotient or ratio) of a matrix and a vector.

See Planetmath for more information.

**RayleighQuotientIteration** `RayleighQuotientIteration (A,x,epsilon,maxiter,vecref)`

Find eigenvalues of `A` using the Rayleigh quotient iteration method. `x` is a guess at a eigenvector and could be random. It should have nonzero imaginary part if it will have any chance at finding complex eigenvalues. The code will run at most `maxiter` iterations and return `null` if we cannot get within an error of `epsilon`. `vecref` should either be `null` or a reference to a variable where the eigenvector should be stored.

See Planetmath for more information on Rayleigh quotient.

**Rank** `Rank (M)`

Aliases: `rank`

Get the rank of a matrix.

See Planetmath for more information.

**RosserMatrix** `RosserMatrix ()`

Returns the Rosser matrix, which is a classic symmetric eigenvalue test problem.

**Rotation2D** `Rotation2D (angle)`

Aliases: `RotationMatrix`

Return the matrix corresponding to rotation around origin in $R^2$.

**Rotation3DX** `Rotation3DX (angle)`

Return the matrix corresponding to rotation around origin in $R^3$ about the x-axis.

**Rotation3DY** `Rotation3DY (angle)`

Return the matrix corresponding to rotation around origin in $R^3$ about the y-axis.

**Rotation3DZ** `Rotation3DZ (angle)`

Return the matrix corresponding to rotation around origin in $R^3$ about the z-axis.

**RowSpace** `RowSpace (M)`

Get a basis matrix for the rowspace of a matrix.

**SesquilinearForm** `SesquilinearForm (v,A,w)`

Evaluate (v,w) with respect to the sesquilinear form given by the matrix A.

**SesquilinearFormFunction** `SesquilinearFormFunction (A)`

Return a function that evaluates two vectors with respect to the sesquilinear form given by A.

**SmithNormalFormField** `SmithNormalFormField (A)`

Returns the Smith normal form of a matrix over fields (will end up with 1's on the diagonal).

See Wikipedia for more information.

**SmithNormalFormInteger** `SmithNormalFormInteger (M)`

Return the Smith normal form for square integer matrices over integers (not its characteristic).

See Wikipedia for more information.

**SolveLinearSystem** `SolveLinearSystem (M,V,args...)`

Solve linear system Mx=V, return solution V if there is a unique solution, `null` otherwise. Extra two reference parameters can optionally be used to get the reduced M and V.

**ToeplitzMatrix** `ToeplitzMatrix (c, r...)`

Return the Toeplitz matrix constructed given the first column c and (optionally) the first row r. If only the column c is given then it is conjugated and the nonconjugated version is used for the first row to give a Hermitian matrix (if the first element is real of course).

See Wikipedia or Planetmath for more information.

**Trace** `Trace (M)`

Aliases: `trace`

Calculate the trace of a matrix. That is the sum of the diagonal elements.

See Wikipedia or Planetmath for more information.

**Transpose** `Transpose (M)`

Transpose of a matrix. This is the same as the `.'` operator.

See Wikipedia or Planetmath for more information.

**VandermondeMatrix** `VandermondeMatrix (v)`

Aliases: `vander`

Return the Vandermonde matrix.

See Wikipedia for more information.

**VectorAngle** `VectorAngle (v,w,B...)`

The angle of two vectors with respect to inner product given by `B`. If `B` is not given then the standard Hermitian product is used. `B` can either be a sesquilinear function of two arguments or it can be a matrix giving a sesquilinear form.

**VectorSpaceDirectSum** `VectorSpaceDirectSum (M,N)`

The direct sum of the vector spaces M and N.

**VectorSubspaceIntersection** `VectorSubspaceIntersection (M,N)`

Intersection of the subspaces given by M and N.

**VectorSubspaceSum** `VectorSubspaceSum (M,N)`

The sum of the vector spaces M and N, that is {w | w=m+n, m in M, n in N}.

**adj** `adj (m)`

Aliases: `Adjugate`

Get the classical adjoint (adjugate) of a matrix.

**cref** `cref (M)`

Aliases: `CREF ColumnReducedEchelonForm`

Compute the Column Reduced Echelon Form.

**det** `det (M)`

Aliases: `Determinant`

Get the determinant of a matrix.

See Wikipedia or Planetmath for more information.

**ref** `ref (M)`

Aliases: `REF RowEchelonForm`

Get the row echelon form of a matrix. That is, apply gaussian elimination but not backaddition to `M`. The pivot rows are divided to make all pivots 1.

See Wikipedia or Planetmath for more information.

**rref** `rref (M)`

Aliases: `RREF ReducedRowEchelonForm`

Get the reduced row echelon form of a matrix. That is, apply gaussian elimination together with backaddition to `M`.

See Wikipedia or Planetmath for more information.

## 11.10 Combinatorics

**Catalan** `Catalan (n)`

Get `n`th Catalan number.

See Planetmath for more information.

**Combinations** `Combinations (k,n)`

Get all combinations of k numbers from 1 to n as a vector of vectors. (See also NextCombination)

**DoubleFactorial** `DoubleFactorial (n)`

Double factorial: **`n(n-2)(n-4)...`**

See Planetmath for more information.

**Factorial** `Factorial (n)`

Factorial: **`n(n-1)(n-2)...`**

See Planetmath for more information.

**FallingFactorial** `FallingFactorial (n,k)`

Falling factorial: **`(n)_k = n(n-1)...(n-(k-1))`**

See Planetmath for more information.

**Fibonacci** `Fibonacci (x)`

Aliases: `fib`

Calculate `n`th Fibonacci number. That is the number defined recursively by **`Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)`** and **`Fibonacci(1) = Fibonacci(2) = 1`**.

See Wikipedia or Planetmath or Mathworld for more information.

**FrobeniusNumber** `FrobeniusNumber (v,arg...)`

Calculate the Frobenius number. That is calculate smallest number that cannot be given as a non-negative integer linear combination of a given vector of non-negative integers. The vector can be given as separate numbers or a single vector. All the numbers given should have GCD of 1.

See Mathworld for more information.

**GaloisMatrix** `GaloisMatrix (combining_rule)`

Galois matrix given a linear combining rule (a_1*x_+...+a_n*x_n=x_(n+1)).

**GreedyAlgorithm** `FrobeniusNumber (n,v)`

Find the vector `c` of non-negative integers such that taking the dot product with `v` is equal to `n`. If not possible returns `null`. `v` should be given sorted in increasing order and should consist of non-negative integers.

See Mathworld for more information.

**HarmonicNumber** `HarmonicNumber (n,r)`

Aliases: `HarmonicH`

Harmonic Number, the `n`th harmonic number of order `r`.

**Hofstadter** `Hofstadter (n)`

Hofstadter's function q(n) defined by q(1)=1, q(2)=1, q(n)=q(n-q(n-1))+q(n-q(n-2)).

**LinearRecursiveSequence** `LinearRecursiveSequence (seed_values,combining_rule,n)`

Compute linear recursive sequence using Galois stepping.

**Multinomial** `Multinomial (v,arg...)`

Calculate multinomial coefficients. Takes a vector of `k` non-negative integers and computes the multinomial coefficient. This corresponds to the coefficient in the homogeneous polynomial in `k` variables with the corresponding powers.

The formula for **`Multinomial(a,b,c)`** can be written as:

```
(a+b+c)! / (a!b!c!)
```

In other words, if we would have only two elements, then **`Multinomial(a,b)`** is the same thing as **`Binomial(a+b,a)`** or **`Binomial(a+b,b)`**.

See Planetmath, Mathworld, or Wikipedia for more information.

**NextCombination** `NextCombination (v,n)`

Get combination that would come after `v` in call to combinations, first combination should be **`[1:k]`**. This function is useful if you have many combinations to go through and you don't want to waste memory to store them all.

For example with Combination you would normally write a loop like:

```
for n in Combinations (4,6) do (
  SomeFunction (n)
);
```

But with NextCombination you would write something like:

```
n:=[1:4];
do (
  SomeFunction (n)
) while not IsNull(n:=NextCombination(n,6));
```

See also Combinations.

**Pascal** `Pascal (i)`

Get the Pascal's triangle as a matrix. This will return an `i`+1 by `i`+1 lower diagonal matrix that is the Pascal's triangle after `i` iterations.

See Planetmath for more information.

**Permutations** `Permutations (k,n)`

Get all permutations of `k` numbers from 1 to `n` as a vector of vectors.

See Mathworld or Wikipedia for more information.

**RisingFactorial** `RisingFactorial (n,k)`

Aliases: `Pochhammer`

(Pochhammer) Rising factorial: (n)_k = n(n+1)...(n+(k-1)).

See Planetmath for more information.

**StirlingNumberFirst** `StirlingNumberFirst (n,m)`

Aliases: `StirlingS1`

Stirling number of the first kind.

See Planetmath or Mathworld for more information.

**StirlingNumberSecond** `StirlingNumberSecond (n,m)`

Aliases: `StirlingS2`

Stirling number of the second kind.

See Planetmath or Mathworld for more information.

**Subfactorial** `Subfactorial (n)`

Subfactorial: n! times sum_{k=1}^n (-1)^k/k!.

**Triangular** `Triangular (nth)`

Calculate the `n`th triangular number.

See Planetmath for more information.

**nCr** `nCr (n,r)`

Aliases: `Binomial`

Calculate combinations, that is, the binomial coefficient. `n` can be any real number.

See Planetmath for more information.

**nPr** `nPr (n,r)`

Calculate the number of permutations of size `r`of numbers from 1 to `n`.

See Mathworld or Wikipedia for more information.

## 11.11 Calculus

**CompositeSimpsonsRule** `CompositeSimpsonsRule (f,a,b,n)`

Integration of f by Composite Simpson's Rule on the interval [a,b] with n subintervals with error of max(f"")*h^4*(b-a)/180, note that n should be even.

See Planetmath for more information.

**CompositeSimpsonsRuleTolerance** `CompositeSimpsonsRuleTolerance (f,a,b,FourthDerivativeBound,Toler`

Integration of f by Composite Simpson's Rule on the interval [a,b] with the number of steps calculated by the fourth derivative bound and the desired tolerance.

See Planetmath for more information.

**Derivative** `Derivative (f,x0)`

Attempt to calculate derivative by trying first symbolically and then numerically.

**EvenPeriodicExtension** `EvenPeriodicExtension (f,L)`

Return a function that is the even periodic extension of `f` with half period `L`. That is a function defined on the interval **`[0,L]`** extended to be even on **`[-L,L]`** and then extended to be periodic with period **`2*L`**.

See also OddPeriodicExtension and PeriodicExtension.

Version 1.0.7 onwards.

**FourierSeriesFunction** `FourierSeriesFunction (a,b,L)`

Return a function that is a Fourier series with the coefficients given by the vectors `a` (sines) and `b` (cosines). Note that **`a@(1)`** is the constant coefficient! That is, **`a@(n)`** refers to the term **`cos(x*(n-1)*pi/L)`**, while **`b@(n)`** refers to the term **`sin(x*n*pi/L)`**. Either `a` or `b` can be `null`.

See Wikipedia or Mathworld for more information.

**InfiniteProduct** `InfiniteProduct (func,start,inc)`

Try to calculate an infinite product for a single parameter function.

**InfiniteProduct2** `InfiniteProduct2 (func,arg,start,inc)`

Try to calculate an infinite product for a double parameter function with func(arg,n).

**InfiniteSum** `InfiniteSum (func,start,inc)`

Try to calculate an infinite sum for a single parameter function.

**InfiniteSum2** `InfiniteSum2 (func,arg,start,inc)`

Try to calculate an infinite sum for a double parameter function with func(arg,n).

**IsContinuous** `IsContinuous (f,x0)`

Try and see if a real-valued function is continuous at x0 by calculating the limit there.

**IsDifferentiable** `IsDifferentiable (f,x0)`

Test for differentiability by approximating the left and right limits and comparing.

**LeftLimit** `LeftLimit (f,x0)`

Calculate the left limit of a real-valued function at x0.

**Limit** `Limit (f,x0)`

Calculate the limit of a real-valued function at x0. Tries to calculate both left and right limits.

**MidpointRule** `MidpointRule (f,a,b,n)`

Integration by midpoint rule.

**NumericalDerivative** `NumericalDerivative (f,x0)`

Aliases: `NDerivative`

Attempt to calculate numerical derivative.

**NumericalFourierSeriesCoefficients** `NumericalFourierSeriesCoefficients (f,L,N)`

Return a vector of vectors **`[a,b]`** where `a` are the cosine coefficients and `b` are the sine coefficients of the Fourier series of `f` with half-period `L` (that is defined on **`[-L,L]`** and extended periodically) with coefficients up to `N`th harmonic computed numerically. The coefficients are computed by numerical integration using NumericalIntegral.

See Wikipedia or Mathworld for more information.

Version 1.0.7 onwards.

**NumericalFourierSeriesFunction** `NumericalFourierSeriesFunction (f,L,N)`

Return a function that is the Fourier series of `f` with half-period `L` (that is defined on **`[-L,L]`** and extended periodically) with coefficients up to `N`th harmonic computed numerically. This is the trigonometric real series composed of sines and cosines. The coefficients are computed by numerical integration using NumericalIntegral.

See Wikipedia or Mathworld for more information.

Version 1.0.7 onwards.

**NumericalFourierCosineSeriesCoefficients** `NumericalFourierCosineSeriesCoefficients (f,L,N)`

Return a vector of coefficients of the cosine Fourier series of `f` with half-period `L`. That is, we take `f` defined on **`[0,L]`** take the even periodic extension and compute the Fourier series, which only has cosine terms. The series is computed up to the `N`th harmonic. The coefficients are computed by numerical integration using NumericalIntegral. Note that **`a@(1)`** is the constant coefficient! That is, **`a@(n)`** refers to the term **`cos(x*(n-1)*pi/L)`**.

See Wikipedia or Mathworld for more information.

Version 1.0.7 onwards.

**NumericalFourierCosineSeriesFunction** `NumericalFourierCosineSeriesFunction (f,L,N)`

Return a function that is the cosine Fourier series of `f` with half-period `L`. That is, we take `f` defined on **`[0,L]`** take the even periodic extension and compute the Fourier series, which only has cosine terms. The series is computed up to the `N`th harmonic. The coefficients are computed by numerical integration using NumericalIntegral.

See Wikipedia or Mathworld for more information.

Version 1.0.7 onwards.

**NumericalFourierSineSeriesCoefficients** `NumericalFourierSineSeriesCoefficients (f,L,N)`

Return a vector of coefficients of the sine Fourier series of `f` with half-period `L`. That is, we take `f` defined on **`[0,L]`** take the odd periodic extension and compute the Fourier series, which only has sine terms. The series is computed up to the `N`th harmonic. The coefficients are computed by numerical integration using NumericalIntegral.

See Wikipedia or Mathworld for more information.

Version 1.0.7 onwards.

**NumericalFourierSineSeriesFunction** `NumericalFourierSineSeriesFunction (f,L,N)`

Return a function that is the sine Fourier series of `f` with half-period `L`. That is, we take `f` defined on **`[0,L]`** take the odd periodic extension and compute the Fourier series, which only has sine terms. The series is computed up to the `N`th harmonic. The coefficients are computed by numerical integration using NumericalIntegral.

See Wikipedia or Mathworld for more information.

Version 1.0.7 onwards.

**NumericalIntegral** `NumericalIntegral (f,a,b)`

Integration by rule set in NumericalIntegralFunction of f from a to b using NumericalIntegralSteps steps.

**NumericalLeftDerivative** `NumericalLeftDerivative (f,x0)`

Attempt to calculate numerical left derivative.

**NumericalLimitAtInfinity** `NumericalLimitAtInfinity (_f,step_fun,tolerance,successive_for_success,`

Attempt to calculate the limit of f(step_fun(i)) as i goes from 1 to N.

**NumericalRightDerivative** `NumericalRightDerivative (f,x0)`

Attempt to calculate numerical right derivative.

**OddPeriodicExtension** `OddPeriodicExtension (f,L)`

Return a function that is the odd periodic extension of `f` with half period `L`. That is a function defined on the interval **`[0,L]`** extended to be odd on **`[-L,L]`** and then extended to be periodic with period **`2*L`**.

See also EvenPeriodicExtension and PeriodicExtension.

Version 1.0.7 onwards.

**OneSidedFivePointFormula** `OneSidedFivePointFormula (f,x0,h)`

Compute one-sided derivative using five point formula.

**OneSidedThreePointFormula** `OneSidedThreePointFormula (f,x0,h)`

Compute one-sided derivative using three-point formula.

**PeriodicExtension** `PeriodicExtension (f,a,b)`

Return a function that is the periodic extension of `f` defined on the interval **`[a,b]`** and has period **`b-a`**.

See also OddPeriodicExtension and EvenPeriodicExtension.

Version 1.0.7 onwards.

**RightLimit** `RightLimit (f,x0)`

Calculate the right limit of a real-valued function at x0.

**TwoSidedFivePointFormula** `TwoSidedFivePointFormula (f,x0,h)`

Compute two-sided derivative using five-point formula.

**TwoSidedThreePointFormula** `TwoSidedThreePointFormula (f,x0,h)`

Compute two-sided derivative using three-point formula.

## 11.12 Functions

**Argument** `Argument (z)`

Aliases: `Arg arg`

argument (angle) of complex number.

**BesselJ0** `BesselJ0 (x)`

Bessel function of the first kind of order 0. Only implemented for real numbers.

See Wikipedia for more information.

Version 1.0.16 onwards.

**BesselJ1** `BesselJ1 (x)`

Bessel function of the first kind of order 1. Only implemented for real numbers.

See Wikipedia for more information.

Version 1.0.16 onwards.

**BesselJn** `BesselJn (n,x)`

Bessel function of the first kind of order `n`. Only implemented for real numbers.

See Wikipedia for more information.

Version 1.0.16 onwards.

**BesselY0** `BesselY0 (x)`

Bessel function of the second kind of order 0. Only implemented for real numbers.

See Wikipedia for more information.

Version 1.0.16 onwards.

**BesselY1** `BesselY1 (x)`

Bessel function of the second kind of order 1. Only implemented for real numbers.

See Wikipedia for more information.

Version 1.0.16 onwards.

**BesselYn** `BesselYn (n,x)`

Bessel function of the second kind of order `n`. Only implemented for real numbers.

See Wikipedia for more information.

Version 1.0.16 onwards.

**DirichletKernel** `DirichletKernel (n,t)`

Dirichlet kernel of order n.

**DiscreteDelta** `DiscreteDelta (v)`

Returns 1 if and only if all elements are zero.

**ErrorFunction** `ErrorFunction (x)`

Aliases: `erf`

The error function, 2/sqrt(pi) * int_0^x e^(-t^2) dt.

See Planetmath for more information.

**FejerKernel** `FejerKernel (n,t)`

Fejer kernel of order `n` evaluated at `t`

See Planetmath for more information.

**GammaFunction** `GammaFunction (x)`

Aliases: `Gamma`

The Gamma function. Currently only implemented for real values.

See Planetmath or Wikipedia for more information.

**KroneckerDelta** `KroneckerDelta (v)`

Returns 1 if and only if all elements are equal.

**LambertW** `LambertW (x)`

The principal branch of Lambert W function computed for only real values greater than or equal to **`-1/e`**. That is, `LambertW` is the inverse of the expression **`x*e^x`**. Even for real `x` this expression is not one to one and therefore has two branches over **`[-1/e,0)`**. See LambertWm1 for the other real branch.

See Wikipedia for more information.

Version 1.0.18 onwards.

**LambertWm1** `LambertWm1 (x)`

The minus-one branch of Lambert W function computed for only real values greater than or equal to **`-1/e`** and less than 0. That is, `LambertWm1` is the second branch of the inverse of **`x*e^x`**. See LambertW for the principal branch.

See Wikipedia for more information.

**MinimizeFunction** `MinimizeFunction (func,x,incr)`

Find the first value where f(x)=0.

**MoebiusDiskMapping** `MoebiusDiskMapping (a,z)`

Moebius mapping of the disk to itself mapping a to 0.

See Planetmath for more information.

**MoebiusMapping** `MoebiusMapping (z,z2,z3,z4)`

Moebius mapping using the cross ratio taking z2,z3,z4 to 1,0, and infinity respectively.

See Planetmath for more information.

**MoebiusMappingInftyToInfty** `MoebiusMappingInftyToInfty (z,z2,z3)`

Moebius mapping using the cross ratio taking infinity to infinity and z2,z3 to 1 and 0 respectively.

See Planetmath for more information.

**MoebiusMappingInftyToOne** `MoebiusMappingInftyToOne (z,z3,z4)`

Moebius mapping using the cross ratio taking infinity to 1 and z3,z4 to 0 and infinity respectively.

See Planetmath for more information.

**MoebiusMappingInftyToZero** `MoebiusMappingInftyToZero (z,z2,z4)`

Moebius mapping using the cross ratio taking infinity to 0 and z2,z4 to 1 and infinity respectively.

See Planetmath for more information.

**PoissonKernel** `PoissonKernel (r,sigma)`

Poisson kernel on D(0,1) (not normalized to 1, that is integral of this is 2pi).

**PoissonKernelRadius** `PoissonKernelRadius (r,sigma)`

Poisson kernel on D(0,R) (not normalized to 1).

**RiemannZeta** `RiemannZeta (x)`

Aliases: `zeta`

The Riemann zeta function. Currently only implemented for real values.

See Planetmath or Wikipedia for more information.

**UnitStep** `UnitStep (x)`

The unit step function is 0 for x<0, 1 otherwise. This is the integral of the Dirac Delta function. Also called the Heaviside function.

See Wikipedia for more information.

**cis** `cis (x)`

The `cis` function, that is the same as **`cos(x)+1i*sin(x)`**

**deg2rad** `deg2rad (x)`

Convert degrees to radians.

**rad2deg** `rad2deg (x)`

Convert radians to degrees.

**sinc** `sinc (x)`

Calculates the unnormalized sinc function, that is **`sin(x)/x`**. If you want the normalized function call **`sinc(pi*x)`**.

See Wikipedia for more information.

Version 1.0.16 onwards.

## 11.13 Equation Solving

**CubicFormula** `CubicFormula (p)`

Compute roots of a cubic (degree 3) polynomial using the cubic formula. The polynomial should be given as a vector of coefficients. That is **`4*x^3 + 2*x + 1`** corresponds to the vector **`[1,2,0,4]`**. Returns a column vector of the three solutions. The first solution is always the real one as a cubic always has one real solution.

See Planetmath, Mathworld, or Wikipedia for more information.

**EulersMethod** `EulersMethod (f,x0,y0,x1,n)`

Use classical Euler's method to numerically solve y'=f(x,y) for initial `x0`, `y0` going to `x1` with `n` increments, returns `y` at `x1`. Unless you explicitly want to use Euler's method, you should really think about using RungeKutta for solving ODE.

Systems can be solved by just having `y` be a (column) vector everywhere. That is, `y0` can be a vector in which case `f` should take a number `x` and a vector of the same size for the second argument and should return a vector of the same size.

See Mathworld, or Wikipedia for more information.

**EulersMethodFull** `EulersMethodFull (f,x0,y0,x1,n)`

Use classical Euler's method to numerically solve y'=f(x,y) for initial `x0`, `y0` going to `x1` with `n` increments, returns a 2 by **`n+1`** matrix with the `x` and `y` values. Unless you explicitly want to use Euler's method, you should really think about using RungeKuttaFull for solving ODE. Suitable for plugging into LinePlotDrawLine or LinePlotDrawPoints.

Example:

```
genius> LinePlotClear();
genius> line = EulersMethodFull(`(x,y)=y,0,1.0,3.0,50);
genius> LinePlotDrawLine(line,"window","fit","color","blue","legend","Exponential ↩
    growth");
```

Systems can be solved by just having `y` be a (column) vector everywhere. That is, `y0` can be a vector in which case `f` should take a number `x` and a vector of the same size for the second argument and should return a vector of the same size.

The output for a system is still a n by 2 matrix with the second entry being a vector. If you wish to plot the line, make sure to use row vectors, and then flatten the matrix with ExpandMatrix, and pick out the right columns. Example:

```
genius> LinePlotClear();
genius> lines = EulersMethodFull(`(x,y)=[y@(2),-y@(1)],0,[1.0,1.0],10.0,500);
genius> lines = ExpandMatrix(lines);
genius> firstline = lines@(,[1,2]);
genius> secondline = lines@(,[1,3]);
genius> LinePlotWindow = [0,10,-2,2];
genius> LinePlotDrawLine(firstline,"color","blue","legend","First");
genius> LinePlotDrawPoints(secondline,"color","red","thickness",3,"legend","Second");
```

See Mathworld, or Wikipedia for more information.

Version 1.0.10 onwards.

**FindRootBisection** `FindRootBisection (f,a,b,TOL,N)`

Find root of a function using the bisection method. `a` and `b` are the initial guess interval, **`f(a)`** and **`f(b)`** should have opposite signs. `TOL` is the desired tolerance and `N` is the limit on the number of iterations to run, 0 means no limit. The function returns a vector **`[success,value,iteration]`**, where `success` is a boolean indicating success, `value` is the last value computed, and `iteration` is the number of iterations done.

**FindRootFalsePosition** `FindRootFalsePosition (f,a,b,TOL,N)`

Find root of a function using the method of false position. `a` and `b` are the initial guess interval, **`f(a)`** and **`f(b)`** should have opposite signs. `TOL` is the desired tolerance and `N` is the limit on the number of iterations to run, 0 means no limit. The function returns a vector **`[success,value,iteration]`**, where `success` is a boolean indicating success, `value` is the last value computed, and `iteration` is the number of iterations done.

**FindRootMullersMethod** `FindRootMullersMethod (f,x0,x1,x2,TOL,N)`

Find root of a function using the Muller's method. `TOL` is the desired tolerance and `N` is the limit on the number of iterations to run, 0 means no limit. The function returns a vector **`[success,value,iteration]`**, where `success` is a boolean indicating success, `value` is the last value computed, and `iteration` is the number of iterations done.

**FindRootSecant** `FindRootSecant (f,a,b,TOL,N)`

Find root of a function using the secant method. `a` and `b` are the initial guess interval, **`f(a)`** and **`f(b)`** should have opposite signs. `TOL` is the desired tolerance and `N` is the limit on the number of iterations to run, 0 means no limit. The function returns a vector **`[success,value,iteration]`**, where `success` is a boolean indicating success, `value` is the last value computed, and `iteration` is the number of iterations done.

**HalleysMethod** `HalleysMethod (f,df,ddf,guess,epsilon,maxn)`

Find zeros using Halleys's method. `f` is the function, `df` is the derivative of `f`, and `ddf` is the second derivative of `f`. `guess` is the initial guess. The function returns after two successive values are within `epsilon` of each other, or after `maxn` tries, in which case the function returns `null` indicating failure.

See also NewtonsMethod and SymbolicDerivative.

Example to find the square root of 10:

```
genius> HalleysMethod(`(x)=x^2-10,`(x)=2*x,`(x)=2,3,10^-10,100)
```

See Wikipedia for more information.

Version 1.0.18 onwards.

**NewtonsMethod** `NewtonsMethod (f,df,guess,epsilon,maxn)`

Find zeros using Newton's method. `f` is the function and `df` is the derivative of `f`. `guess` is the initial guess. The function returns after two successive values are within `epsilon` of each other, or after `maxn` tries, in which case the function returns `null` indicating failure.

See also NewtonsMethodPoly and SymbolicDerivative.

Example to find the square root of 10:

```
genius> NewtonsMethod(`(x)=x^2-10,`(x)=2*x,3,10^-10,100)
```

See Wikipedia for more information.

Version 1.0.18 onwards.

**PolynomialRoots** `PolynomialRoots (p)`

Compute roots of a polynomial (degrees 1 through 4) using one of the formulas for such polynomials. The polynomial should be given as a vector of coefficients. That is **`4*x^3 + 2*x + 1`** corresponds to the vector **`[1,2,0,4]`**. Returns a column vector of the solutions.

The function calls QuadraticFormula, CubicFormula, and QuarticFormula.

**QuadraticFormula** `QuadraticFormula (p)`

Compute roots of a quadratic (degree 2) polynomial using the quadratic formula. The polynomial should be given as a vector of coefficients. That is **`3*x^2 + 2*x + 1`** corresponds to the vector **`[1,2,3]`**. Returns a column vector of the two solutions.

See Planetmath or Mathworld for more information.

**QuarticFormula** `QuarticFormula (p)`

Compute roots of a quartic (degree 4) polynomial using the quartic formula. The polynomial should be given as a vector of coefficients. That is **`5*x^4 + 2*x + 1`** corresponds to the vector **`[1,2,0,0,5]`**. Returns a column vector of the four solutions.

See Planetmath, Mathworld, or Wikipedia for more information.

**RungeKutta** `RungeKutta (f,x0,y0,x1,n)`

Use classical non-adaptive fourth order Runge-Kutta method to numerically solve y'=f(x,y) for initial `x0`, `y0` going to `x1` with `n` increments, returns `y` at `x1`.

Systems can be solved by just having `y` be a (column) vector everywhere. That is, `y0` can be a vector in which case `f` should take a number `x` and a vector of the same size for the second argument and should return a vector of the same size.

See Mathworld, or Wikipedia for more information.

**RungeKuttaFull** `RungeKuttaFull (f,x0,y0,x1,n)`

Use classical non-adaptive fourth order Runge-Kutta method to numerically solve y'=f(x,y) for initial `x0`, `y0` going to `x1` with `n` increments, returns a 2 by **`n+1`** matrix with the `x` and `y` values. Suitable for plugging into LinePlotDrawLine or LinePlotDrawPoints.

Example:

```
genius> LinePlotClear();
genius> line = RungeKuttaFull(`(x,y)=y,0,1.0,3.0,50);
genius> LinePlotDrawLine(line,"window","fit","color","blue","legend","Exponential  ↩
    growth");
```

Systems can be solved by just having `y` be a (column) vector everywhere. That is, `y0` can be a vector in which case `f` should take a number `x` and a vector of the same size for the second argument and should return a vector of the same size.

The output for a system is still a n by 2 matrix with the second entry being a vector. If you wish to plot the line, make sure to use row vectors, and then flatten the matrix with ExpandMatrix, and pick out the right columns. Example:

```
genius> LinePlotClear();
genius> lines = RungeKuttaFull(`(x,y)=[y@(2),-y@(1)],0,[1.0,1.0],10.0,100);
genius> lines = ExpandMatrix(lines);
genius> firstline = lines@(,[1,2]);
genius> secondline = lines@(,[1,3]);
genius> LinePlotWindow = [0,10,-2,2];
genius> LinePlotDrawLine(firstline,"color","blue","legend","First");
genius> LinePlotDrawPoints(secondline,"color","red","thickness",3,"legend","Second");
```

See Mathworld, or Wikipedia for more information.

Version 1.0.10 onwards.

## 11.14 Statistics

**Average** `Average (m)`

Aliases: `average` `Mean` `mean`

Calculate average of an entire matrix.

See Mathworld for more information.

**GaussDistribution** `GaussDistribution (x,sigma)`

Integral of the GaussFunction from 0 to `x` (area under the normal curve).

See Mathworld for more information.

**GaussFunction** `GaussFunction (x,sigma)`

The normalized Gauss distribution function (the normal curve).

See Mathworld for more information.

**Median** `Median (m)`

Aliases: `median`

Calculate median of an entire matrix.

See Mathworld for more information.

**PopulationStandardDeviation** `PopulationStandardDeviation (m)`

Aliases: `stdevp`

Calculate the population standard deviation of a whole matrix.

**RowAverage** `RowAverage (m)`

Aliases: `RowMean`

Calculate average of each row in a matrix.

See Mathworld for more information.

**RowMedian** `RowMedian (m)`

Calculate median of each row in a matrix and return a column vector of the medians.

See Mathworld for more information.

**RowPopulationStandardDeviation** `RowPopulationStandardDeviation (m)`

Aliases: `rowstdevp`

Calculate the population standard deviations of rows of a matrix and return a vertical vector.

**RowStandardDeviation** `RowStandardDeviation (m)`

Aliases: `rowstdev`

Calculate the standard deviations of rows of a matrix and return a vertical vector.

**StandardDeviation** `StandardDeviation (m)`

Aliases: `stdev`

Calculate the standard deviation of a whole matrix.

## 11.15 Polynomials

**AddPoly** `AddPoly (p1,p2)`

Add two polynomials (vectors).

**DividePoly** `DividePoly (p,q,&r)`

Divide two polynomials (as vectors) using long division. Returns the quotient of the two polynomials. The optional argument `r` is used to return the remainder. The remainder will have lower degree than `q`.

See Planetmath for more information.

**IsPoly** `IsPoly (p)`

Check if a vector is usable as a polynomial.

**MultiplyPoly** `MultiplyPoly (p1,p2)`

Multiply two polynomials (as vectors).

**NewtonsMethodPoly** `NewtonsMethodPoly (poly,guess,epsilon,maxn)`

Find a root of a polynomial using Newton's method. `poly` is the polynomial as a vector and `guess` is the initial guess. The function returns after two successive values are within `epsilon` of each other, or after `maxn` tries, in which case the function returns `null` indicating failure.

See also NewtonsMethod.

Example to find the square root of 10:

```
genius> NewtonsMethodPoly([-10,0,1],3,10^-10,100)
```

See Wikipedia for more information.

**Poly2ndDerivative** `Poly2ndDerivative (p)`

Take second polynomial (as vector) derivative.

**PolyDerivative** `PolyDerivative (p)`

Take polynomial (as vector) derivative.

**PolyToFunction** `PolyToFunction (p)`

Make function out of a polynomial (as vector).

**PolyToString** `PolyToString (p,var...)`

Make string out of a polynomial (as vector).

**SubtractPoly** `SubtractPoly (p1,p2)`

Subtract two polynomials (as vectors).

**TrimPoly** `TrimPoly (p)`

Trim zeros from a polynomial (as vector).

## 11.16 Set Theory

**Intersection** `Intersection (X,Y)`

Returns a set theoretic intersection of X and Y (X and Y are vectors pretending to be sets).

**IsIn** `IsIn (x,X)`

Returns `true` if the element x is in the set X (where X is a vector pretending to be a set).

**IsSubset** `IsSubset (X, Y)`

Returns `true` if X is a subset of Y (X and Y are vectors pretending to be sets).

**MakeSet** `MakeSet (X)`

Returns a vector where every element of X appears only once.

**SetMinus** `SetMinus (X,Y)`

Returns a set theoretic difference X-Y (X and Y are vectors pretending to be sets).

**Union** `Union (X,Y)`

Returns a set theoretic union of X and Y (X and Y are vectors pretending to be sets).

## 11.17 Commutative Algebra

**MacaulayBound** `MacaulayBound (c,d)`

For a Hilbert function that is c for degree d, given the Macaulay bound for the Hilbert function of degree d+1 (The c^<d> operator from Green's proof).

Version 1.0.15 onwards.

**MacaulayLowerOperator** `MacaulayLowerOperator (c,d)`

The c_<d> operator from Green's proof of Macaulay's Theorem.

Version 1.0.15 onwards.

**MacaulayRep** `MacaulayRep (c,d)`

Return the dth Macaulay representation of a positive integer c.

Version 1.0.15 onwards.

## 11.18 Miscellaneous

**ASCIIToString** `ASCIIToString (vec)`

Convert a vector of ASCII values to a string.

**AlphabetToString** `AlphabetToString (vec,alphabet)`

Convert a vector of 0-based alphabet values (positions in the alphabet string) to a string.

**StringToASCII** `StringToASCII (str)`

Convert a string to a vector of ASCII values.

**StringToAlphabet** `StringToAlphabet (str,alphabet)`

Convert a string to a vector of 0-based alphabet values (positions in the alphabet string), -1's for unknown letters.

## 11.19 Symbolic Operations

**SymbolicDerivative** `SymbolicDerivative (f)`

Attempt to symbolically differentiate the function f, where f is a function of one variable.

Examples:

```
genius> SymbolicDerivative(sin)
= (`(x)=cos(x))
genius> SymbolicDerivative(`(x)=7*x^2)
= (`(x)=(7*(2*x)))
```

**SymbolicDerivativeTry** `SymbolicDerivativeTry (f)`

Attempt to symbolically differentiate the function f, where f is a function of one variable, returns `null` if unsuccessful but is silent. (See SymbolicDerivative)

**SymbolicNthDerivative** `SymbolicNthDerivative (f,n)`

Attempt to symbolically differentiate a function n times. (See SymbolicDerivative)

**SymbolicNthDerivativeTry** `SymbolicNthDerivativeTry (f,n)`

Attempt to symbolically differentiate a function n times quietly and return `null` on failure (See SymbolicNthDerivative)

**SymbolicTaylorApproximationFunction** `SymbolicTaylorApproximationFunction (f,x0,n)`

Attempt to construct the Taylor approximation function around x0 to the nth degree. (See SymbolicDerivative)

## 11.20 Plotting

**ExportPlot** `ExportPlot (file,type)`

`ExportPlot (file)`

Export the contents of the plotting window to a file. The type is a string that specifies the file type to use, "png", "eps", or "ps". If the type is not specified, then it is taken to be the extension, in which case the extension must be ".png", ".eps", or ".ps".

Note that files are overwritten without asking.

On successful export, true is returned. Otherwise error is printed and exception is raised.

Examples:

```
genius> ExportPlot("file.png")
genius> ExportPlot("/directory/file","eps")
```

Version 1.0.16 onwards.

**LinePlot** `LinePlot (func1,func2,func3,...)`

`LinePlot (func1,func2,func3,x1,x2)`

`LinePlot (func1,func2,func3,x1,x2,y1,y2)`

`LinePlot (func1,func2,func3,[x1,x2])`

`LinePlot (func1,func2,func3,[x1,x2,y1,y2])`

Plot a function (or several functions) with a line. First (up to 10) arguments are functions, then optionally you can specify the limits of the plotting window as `x1`, `x2`, `y1`, `y2`. If limits are not specified, then the currently set limits apply (See LinePlotWindow) If the y limits are not specified, then the functions are computed and then the maxima and minima are used.

The parameter LinePlotDrawLegends controls the drawing of the legend.

Examples:

```
genius> LinePlot(sin,cos)
genius> LinePlot(`(x)=x^2,-1,1,0,1)
```

**LinePlotClear** `LinePlotClear ()`

Show the line plot window and clear out functions and any other lines that were drawn.

**LinePlotDrawLine** `LinePlotDrawLine (x1,y1,x2,y2,...)`

`LinePlotDrawLine (v,...)`

Draw a line from `x1`,`y1` to `x2`,`y2`. `x1`,`y1`, `x2`,`y2` can be replaced by an `n` by 2 matrix for a longer polyline.

Extra parameters can be added to specify line color, thickness, arrows, the plotting window, or legend. You can do this by adding an argument string **`"color"`**, **`"thickness"`**, **`"window"`**, **`"arrow"`**, or **`"legend"`**, and after it specify the color, the thickness, the window as 4-vector, type of arrow, or the legend. (Arrow and window are from version 1.0.6 onwards.)

The color should be either a string indicating the common English word for the color that GTK will recognize such as **`"red"`**, **`"blue"`**, **`"yellow"`**, etc... Alternatively the color can be specified in RGB format as **`"#rgb"`**, **`"#rrggbb"`**, or **`"#rrrrggggbbbb"`**, where the r, g, or b are hex digits of the red, green, and blue components of the color. Finally,

since version 1.0.18, the color can also be specified as a real vector specifying the red green and blue components where the components are between 0 and 1, e.g. **[1.0,0.5,0.1]**.

The window should be given as usual as **[x1,x2,y1,y2]**, or alternatively can be given as a string **"fit"** in which case, the x range will be set precisely and the y range will be set with five percent borders around the line.

Arrow specification should be **"origin"**, **"end"**, **"both"**, or **"none"**.

Finally, legend should be a string that can be used as the legend in the graph. That is, if legends are being printed.

Examples:

```
genius> LinePlotDrawLine(0,0,1,1,"color","blue","thickness",3)
genius> LinePlotDrawLine([0,0;1,-1;-1,-1])
genius> LinePlotDrawLine([0,0;1,1],"arrow","end")
genius> LinePlotDrawLine(RungeKuttaFull(`(x,y)=y,0,3,100),"color","blue","legend","The  ↩
    Solution")
genius> for r=0.0 to 1.0 by 0.1 do LinePlotDrawLine([0,0;1,r],"color",[r,(1-r),0.5]," ↩
    window",[0,1,0,1])
```

**LinePlotDrawPoints** `LinePlotDrawPoints (x,y,...)`

`LinePlotDrawPoints (v,...)`

Draw a point at `x`,`y`. The input can be an `n` by 2 matrix for `n` different points. This function has essentially the same input as LinePlotDrawLine.

Extra parameters can be added to specify line color, thickness, the plotting window, or legend. You can do this by adding an argument string **"color"**, **"thickness"**, **"window"**, or **"legend"**, and after it specify the color, the thickness, the window as 4-vector, or the legend.

The color should be either a string indicating the common English word for the color that GTK will recognize such as **"red"**, **"blue"**, **"yellow"**, etc... Alternatively the color can be specified in RGB format as **"#rgb"**, **"#rrggbb"**, or **"#rrrrggggbbbb"**, where the r, g, or b are hex digits of the red, green, and blue components of the color. Finally the color can also be specified as a real vector specifying the red green and blue components where the components are between 0 and 1.

The window should be given as usual as **[x1,x2,y1,y2]**, or alternatively can be given as a string **"fit"** in which case, the x range will be set precisely and the y range will be set with five percent borders around the line.

Finally, legend should be a string that can be used as the legend in the graph. That is, if legends are being printed.

Examples:

```
genius> LinePlotDrawPoints(0,0,"color","blue","thickness",3)
genius> LinePlotDrawPoints([0,0;1,-1;-1,-1])
genius> LinePlotDrawPoints(RungeKuttaFull(`(x,y)=y,0,3,100),"color","blue","legend"," ↩
    The Solution")
```

Available from version 1.0.18 onwards.

**LinePlotParametric** `LinePlotParametric (xfunc,yfunc,...)`

`LinePlotParametric (xfunc,yfunc,t1,t2,tinc)`

`LinePlotParametric (xfunc,yfunc,t1,t2,tinc,x1,x2,y1,y2)`

`LinePlotParametric (xfunc,yfunc,t1,t2,tinc,[x1,x2,y1,y2])`

`LinePlotParametric (xfunc,yfunc,t1,t2,tinc,"fit")`

Plot a parametric function with a line. First come the functions for `x` and `y` then optionally the `t` limits as **t1,t2,tinc**, then optionally the limits as **x1,x2,y1,y2**.

If x and y limits are not specified, then the currently set limits apply (See LinePlotWindow). If instead the string "fit" is given for the x and y limits, then the limits are the maximum extent of the graph

The parameter LinePlotDrawLegends controls the drawing of the legend.

**LinePlotCParametric** `LinePlotCParametric (func,...)`

`LinePlotCParametric (func,t1,t2,tinc)`

`LinePlotCParametric (func,t1,t2,tinc,x1,x2,y1,y2)`

Plot a parametric complex valued function with a line. First comes the function that returns `x+iy`, then optionally the `t` limits as **`t1,t2,tinc`**, then optionally the limits as **`x1,x2,y1,y2`**.

If limits are not specified, then the currently set limits apply (See LinePlotWindow). If instead the string "fit" is given for the x and y limits, then the limits are the maximum extent of the graph

The parameter LinePlotDrawLegends controls the drawing of the legend.

**PlotCanvasFreeze** `PlotCanvasFreeze ()`

Freeze drawing of the canvas plot temporarily. Useful if you need to draw a bunch of elements and want to delay drawing everything to avoid flicker in an animation. After everything has been drawn you should call PlotCanvasThaw.

The canvas is always thawed after end of any execution, so it will never remain frozen. The moment a new command line is shown for example the plot canvas is thawed automatically. Also note that calls to freeze and thaw may be safely nested.

Version 1.0.18 onwards.

**PlotCanvasThaw** `PlotCanvasThaw ()`

Thaw the plot canvas frozen by PlotCanvasFreeze and redraw the canvas immediately. The canvas is also always thawed after end of execution of any program.

Version 1.0.18 onwards.

**PlotWindowPresent** `PlotWindowPresent ()`

Show and raise the plot window, creating it if necessary. Normally the window is created when one of the plotting functions is called, but it is not always raised if it happens to be below other windows. So this function is good to call in scripts where the plot window might have been created before, and by now hidden behind the console or other windows.

Version 1.0.19 onwards.

**SlopefieldClearSolutions** `SlopefieldClearSolutions ()`

Clears the solutions drawn by the SlopefieldDrawSolution function.

**SlopefieldDrawSolution** `SlopefieldDrawSolution (x, y, dx)`

When a slope field plot is active, draw a solution with the specified initial condition. The standard Runge-Kutta method is used with increment `dx`. Solutions stay on the graph until a different plot is shown or until you call SlopefieldClearSolutions. You can also use the graphical interface to draw solutions and specify initial conditions with the mouse.

**SlopefieldPlot** `SlopefieldPlot (func)`

`SlopefieldPlot (func,x1,x2,y1,y2)`

Plot a slope field. The function `func` should take two real numbers `x` and `y`, or a single complex number. Optionally you can specify the limits of the plotting window as `x1`, `x2`, `y1`, `y2`. If limits are not specified, then the currently set limits apply (See LinePlotWindow).

The parameter LinePlotDrawLegends controls the drawing of the legend.

Examples:

```
genius> Slopefield(`(x,y)=sin(x-y),-5,5,-5,5)
```

**SurfacePlot** `SurfacePlot (func)`

`SurfacePlot (func,x1,x2,y1,y2,z1,z2)`

```
SurfacePlot (func,x1,x2,y1,y2)

SurfacePlot (func,[x1,x2,y1,y2,z1,z2])

SurfacePlot (func,[x1,x2,y1,y2])
```

Plot a surface function that takes either two arguments or a complex number. First comes the function then optionally limits as `x1`, `x2`, `y1`, `y2`, `z1`, `z2`. If limits are not specified, then the currently set limits apply (See SurfacePlotWindow). Genius can only plot a single surface function at this time.

If the z limits are not specified then the maxima and minima of the function are used.

Examples:

```
genius> SurfacePlot(|sin|,-1,1,-1,1,0,1.5)
genius> SurfacePlot(`(x,y)=x^2+y,-1,1,-1,1,-2,2)
genius> SurfacePlot(`(z)=|z|^2,-1,1,-1,1,0,2)
```

**SurfacePlotClear** `SurfacePlotClear ()`

Show the surface plot window and clear out functions and any other lines that were drawn.

Available in version 1.0.19 and onwards.

**SurfacePlotData** `SurfacePlotData (data)`

```
SurfacePlotData (data,label)

SurfacePlotData (data,x1,x2,y1,y2,z1,z2)

SurfacePlotData (data,label,x1,x2,y1,y2,z1,z2)

SurfacePlotData (data,[x1,x2,y1,y2,z1,z2])

SurfacePlotData (data,label,[x1,x2,y1,y2,z1,z2])
```

Plot a surface from data. The data is an n by 3 matrix whose rows are the x, y and z coordinates. The data can also be simply a vector whose length is a multiple of 3 and so contains the triples of x, y, z. The data should contain at least 3 points.

Optionally we can give the label and also optionally the limits. If limits are not given, they are computed from the data, SurfacePlotWindow is not used, if you want to use it, pass it in explicitly. If label is not given then empty label is used.

Examples:

```
genius> SurfacePlotData([0,0,0;1,0,1;0,1,1;1,1,3])
genius> SurfacePlotData(data,"My data")
genius> SurfacePlotData(data,-1,1,-1,1,0,10)
genius> SurfacePlotData(data,SurfacePlotWindow)
```

Here's an example of how to plot in polar coordinates, in particular how to plot the function **`-r^2 * theta`**:

```
genius> d:=null; for r=0 to 1 by 0.1 do for theta=0 to 2*pi by pi/5 do d=[d;[r*cos( ↩
    theta),r*sin(theta),-r^2*theta]];
genius> SurfacePlotData(d)
```

Version 1.0.16 onwards.

**SurfacePlotDataGrid** `SurfacePlotDataGrid (data,[x1,x2,y1,y2])`

```
SurfacePlotDataGrid (data,[x1,x2,y1,y2,z1,z2])
```

`SurfacePlotDataGrid (data,[x1,x2,y1,y2],label)`

`SurfacePlotDataGrid (data,[x1,x2,y1,y2,z1,z2],label)`

Plot a surface from regular rectangular data. The data is given in a n by m matrix where the rows are the x coordinate and the columns are the y coordinate. The x coordinate is divided into equal n-1 subintervals and y coordinate is divided into equal m-1 subintervals. The limits `x1` and `x2` give the interval on the x-axis that we use, and the limits `y1` and `y2` give the interval on the y-axis that we use. If the limits `z1` and `z2` are not given they are computed from the data (to be the extreme values from the data).

Optionally we can give the label, if label is not given then empty label is used.

Examples:

```
genius> SurfacePlotDataGrid([1,2;3,4],[0,1,0,1])
genius> SurfacePlotDataGrid(data,[-1,1,-1,1],"My data")
genius> d:=null; for i=1 to 20 do for j=1 to 10 d@(i,j) = (0.1*i-1)^2-(0.1*j)^2;
genius> SurfacePlotDataGrid(d,[-1,1,0,1],"half a saddle")
```

Version 1.0.16 onwards.

**SurfacePlotDrawLine** `SurfacePlotDrawLine (x1,y1,z1,x2,y2,z2,...)`

`SurfacePlotDrawLine (v,...)`

Draw a line from `x1`,`y1`,`z1` to `x2`,`y2`,`z2`. `x1`,`y1`,`z1`, `x2`,`y2`,`z2` can be replaced by an `n` by 3 matrix for a longer polyline.

Extra parameters can be added to specify line color, thickness, arrows, the plotting window, or legend. You can do this by adding an argument string **`"color"`**, **`"thickness"`**, **`"window"`**, or **`"legend"`**, and after it specify the color, the thickness, the window as 6-vector, or the legend.

The color should be either a string indicating the common English word for the color that GTK will recognize such as **`"red"`**, **`"blue"`**, **`"yellow"`**, etc... Alternatively the color can be specified in RGB format as **`"#rgb"`**, **`"#rrggbb"`**, or **`"#rrrrggggbbbb"`**, where the r, g, or b are hex digits of the red, green, and blue components of the color. Finally, since version 1.0.18, the color can also be specified as a real vector specifying the red green and blue components where the components are between 0 and 1, e.g. **`[1.0,0.5,0.1]`**.

The window should be given as usual as **`[x1,x2,y1,y2,z1,z2]`**, or alternatively can be given as a string **`"fit"`** in which case, the x range will be set precisely and the y range will be set with five percent borders around the line.

Finally, legend should be a string that can be used as the legend in the graph. That is, if legends are being printed.

Examples:

```
genius> SurfacePlotDrawLine(0,0,0,1,1,1,"color","blue","thickness",3)
genius> SurfacePlotDrawLine([0,0,0;1,-1,2;-1,-1,-3])
```

Available from version 1.0.19 onwards.

**SurfacePlotDrawPoints** `SurfacePlotDrawPoints (x,y,z,...)`

`SurfacePlotDrawPoints (v,...)`

Draw a point at `x`,`y`,`z`. The input can be an `n` by 3 matrix for `n` different points. This function has essentially the same input as SurfacePlotDrawLine.

Extra parameters can be added to specify line color, thickness, the plotting window, or legend. You can do this by adding an argument string **`"color"`**, **`"thickness"`**, **`"window"`**, or **`"legend"`**, and after it specify the color, the thickness, the window as 6-vector, or the legend.

The color should be either a string indicating the common English word for the color that GTK will recognize such as **`"red"`**, **`"blue"`**, **`"yellow"`**, etc... Alternatively the color can be specified in RGB format as **`"#rgb"`**, **`"#rrggbb"`**, or **`"#rrrrggggbbbb"`**, where the r, g, or b are hex digits of the red, green, and blue components of the color. Finally

the color can also be specified as a real vector specifying the red green and blue components where the components are between 0 and 1.

The window should be given as usual as **`[x1,x2,y1,y2,z1,z2]`**, or alternatively can be given as a string **`"fit"`** in which case, the x range will be set precisely and the y range will be set with five percent borders around the line.

Finally, legend should be a string that can be used as the legend in the graph. That is, if legends are being printed.

Examples:

```
genius> SurfacePlotDrawPoints(0,0,0,"color","blue","thickness",3)
genius> SurfacePlotDrawPoints([0,0,0;1,-1,2;-1,-1,1])
```

Available from version 1.0.19 onwards.

**VectorfieldClearSolutions** `VectorfieldClearSolutions ()`

Clears the solutions drawn by the VectorfieldDrawSolution function.

Version 1.0.6 onwards.

**VectorfieldDrawSolution** `VectorfieldDrawSolution (x, y, dt, tlen)`

When a vector field plot is active, draw a solution with the specified initial condition. The standard Runge-Kutta method is used with increment `dt` for an interval of length `tlen`. Solutions stay on the graph until a different plot is shown or until you call VectorfieldClearSolutions. You can also use the graphical interface to draw solutions and specify initial conditions with the mouse.

Version 1.0.6 onwards.

**VectorfieldPlot** `VectorfieldPlot (funcx, funcy)`

`VectorfieldPlot (funcx, funcy, x1, x2, y1, y2)`

Plot a two dimensional vector field. The function `funcx` should be the dx/dt of the vectorfield and the function `funcy` should be the dy/dt of the vectorfield. The functions should take two real numbers `x` and `y`, or a single complex number. When the parameter VectorfieldNormalized is `true`, then the magnitude of the vectors is normalized. That is, only the direction and not the magnitude is shown.

Optionally you can specify the limits of the plotting window as `x1`, `x2`, `y1`, `y2`. If limits are not specified, then the currently set limits apply (See LinePlotWindow).

The parameter LinePlotDrawLegends controls the drawing of the legend.

Examples:

```
genius> VectorfieldPlot(`(x,y)=x^2-y, `(x,y)=y^2-x, -1, 1, -1, 1)
```

# Chapter 12

# Example Programs in GEL

Here is a function that calculates factorials:

```
function f(x) = if x <= 1 then 1 else (f(x-1)*x)
```

With indentation it becomes:

```
function f(x) = (
  if x <= 1 then
    1
  else
    (f(x-1)*x)
)
```

This is a direct port of the factorial function from the bc manpage. The syntax seems similar to bc, but different in that in GEL, the last expression is the one that is returned. Using the `return` function instead, it would be:

```
function f(x) = (
  if (x <= 1) then return (1);
  return (f(x-1) * x)
)
```

By far the easiest way to define a factorial function would be using the product loop as follows. This is not only the shortest and fastest, but also probably the most readable version.

```
function f(x) = prod k=1 to x do k
```

Here is a larger example, this basically redefines the internal ref function to calculate the row echelon form of a matrix. The function `ref` is built in and much faster, but this example demonstrates some of the more complex features of GEL.

```
# Calculate the row-echelon form of a matrix
function MyOwnREF(m) = (
  if not IsMatrix(m) or not IsValueOnly(m) then
    (error("MyOwnREF: argument not a value only matrix");bailout);
  s := min(rows(m), columns(m));
  i := 1;
  d := 1;
  while d <= s and i <= columns(m) do (

    # This just makes the anchor element non-zero if at
    # all possible
    if m@(d,i) == 0 then (
      j := d+1;
      while j <= rows(m) do (
        if m@(j,i) == 0 then
```

```
          (j=j+1;continue);
        a := m@(j,);
        m@(j,) := m@(d,);
        m@(d,) := a;
        j := j+1;
        break
      )
    );
    if m@(d,i) == 0 then
      (i:=i+1;continue);

    # Here comes the actual zeroing of all but the anchor
    # element rows
    j := d+1;
    while j <= rows(m)) do (
      if m@(j,i) != 0 then (
        m@(j,) := m@(j,)-(m@(j,i)/m@(d,i))*m@(d,)
      );
      j := j+1
    );
    m@(d,) := m@(d,) * (1/m@(d,i));
    d := d+1;
    i := i+1
  );
  m
)
```

# Chapter 13

# Settings

To configure Genius Mathematics Tool, choose Settings → Preferences. There are several basic parameters provided by the calculator in addition to the ones provided by the standard library. These control how the calculator behaves.

**Changing Settings with GEL**
Many of the settings in Genius are simply global variables, and can be evaluated and assigned to in the same way as normal variables. See Section 5.2 about evaluating and assigning to variables, and Section 11.3 for a list of settings that can be modified in this way.
As an example, you can set the maximum number of digits in a result to 12 by typing:

```
MaxDigits = 12
```

## 13.1 Output

**Maximum digits to output** The maximum digits in a result (MaxDigits)

**Results as floats** If the results should be always printed as floats (ResultsAsFloats)

**Floats in scientific notation** If floats should be in scientific notation (ScientificNotation)

**Always print full expressions** Should we print out full expressions for non-numeric return values (longer than a line) (FullExpressions)

**Use mixed fractions** If fractions should be printed as mixed fractions such as "1 1/3" rather than "4/3". (MixedFractions)

**Display 0.0 when floating point number is less than 10^-x (0=never chop)** How to chop output. But only when other numbers nearby are large. See the documentation of the parameter OutputChopExponent.

**Only chop numbers when another number is greater than 10^-x** When to chop output. This is set by the parameter OutputChopWhenExponent. See the documentation of the parameter OutputChopExponent.

**Remember output settings across sessions** Should the output settings in the Number/Expression output options frame be remembered for next session. Does not apply to the Error/Info output options frame.

If unchecked, either the default or any previously saved settings are used each time Genius starts up. Note that settings are saved at the end of the session, so if you wish to change the defaults check this box, restart Genius Mathematics Tool and then uncheck it again.

**Display errors in a dialog** If set the errors will be displayed in a separate dialog, if unset the errors will be printed on the console.

**Display information messages in a dialog** If set the information messages will be displayed in a separate dialog, if unset the information messages will be printed on the console.

**Maximum errors to display** The maximum number of errors to return on one evaluation (MaxErrors). If you set this to 0 then all errors are always returned. Usually if some loop causes many errors, then it is unlikely that you will be able to make sense out of more than a few of these, so seeing a long list of errors is usually not helpful.

In addition to these preferences, there are some preferences that can only be changed by setting them in the workspace console. For others that may affect the output see Section 11.3.

**`IntegerOutputBase`** The base that will be used to output integers

**`OutputStyle`** A string, can be `"normal"`, `"latex"`, `"mathml"` or `"troff"` and it will affect how matrices (and perhaps other stuff) is printed, useful for pasting into documents. Normal style is the default human readable printing style of Genius Mathematics Tool. The other styles are for typsetting in LaTeX, MathML (XML), or in Troff.

## 13.2 Precision

**Floating point precision** The floating point precision in bits (FloatPrecision). Note that changing this only affects newly computed quantities. Old values stored in variables are obviously still in the old precision and if you want to have them more precise you will have to recompute them. Exceptions to this are the system constants such as pi or e.

**Remember precision setting across sessions** Should the precision setting be remembered for the next session. If unchecked, either the default or any previously saved setting is used each time Genius starts up. Note that settings are saved at the end of the session, so if you wish to change the default check this box, restart genius and then uncheck it again.

## 13.3 Terminal

Terminal refers to the console in the work area.

**Scrollback lines** Lines of scrollback in the terminal.

**Font** The font to use on the terminal.

**Black on white** If to use black on white on the terminal.

**Blinking cursor** If the cursor in the terminal should blink when the terminal is in focus. This can sometimes be annoying and it generates idle traffic if you are using Genius remotely.

## 13.4 Memory

**Maximum number of nodes to allocate** Internally all data is put onto small nodes in memory. This gives a limit on the maximum number of nodes to allocate for computations. This limit avoids the problem of running out of memory if you do something by mistake that uses too much memory, such as a recursion without end. This could slow your computer and make it hard to even interrupt the program.

Once the limit is reached, Genius Mathematics Tool asks if you wish to interrupt the computation or if you wish to continue. If you continue, no limit is applied and it will be possible to run your computer out of memory. The limit will be applied again next time you execute a program or an expression on the Console regardless of how you answered the question.

Setting the limit to zero means there is no limit to the amount of memory that genius uses.

# Chapter 14

# About Genius Mathematics Tool

Genius Mathematics Tool was written by Jiří (George) Lebl (jirka@5z.com). The history of Genius Mathematics Tool goes back to late 1997. It was the first calculator program for GNOME, but it then grew beyond being just a desktop calculator. To find more information about Genius Mathematics Tool, please visit the Genius Web page.

To report a bug or make a suggestion regarding this application or this manual, send email to me (the author) or post to the mailing list (see the web page).

This program is distributed under the terms of the GNU General Public license as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version. A copy of this license can be found at this link, or in the file COPYING included with the source code of this program.

Jiří Lebl was during various parts of the development partially supported for the work by NSF grants DMS 0900885, DMS 1362337, the University of Illinois at Urbana-Champaign, the University of California at San Diego, the University of Wisconsin-Madison, and Oklahoma State University. The software has been used for both teaching and research.